UNIVERSITÀ DEGLI STUDI DI MODENA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elet-Designer: uno strumento intelligente orientato agli oggetti per la progettazione di impianti elettrici industriali

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Stefano Riccio

Correlatore
Dott. Ing. Domenico Beneventano

Anno Accademico 1996 - 97

Parole chiave:

Basi di dati ad oggetti
Basi di dati relazionali
Controllo di consistenza
Progetto concettuale
Impianti elettrici industriali

RINGRAZIAMENTI

Nell'ambito accademico desidero ringraziare il mio relatore, **Prof.ssa Sonia Bergamaschi**, per la sua grande disponibilita', l'**Ing. Domenico Beneventano** per l'aiuto alla stesura della parte teorica, l'**Ing. Maurizio Vincini** e la **Dott.ssa Alessandra Garuti** per l'aiuto nei problemi tecnici.
Nell'ambito della A.S.T. System Automation, intendo ringraziare il titolare **Sig. Aristide Stradi**, per la fiducia concordatomi in questo progetto e tutti i colleghi di lavoro, in particolare **Marcello Righi** per l'indispensabile supporto implementativo.
Infine un grande grazie alla mia famiglia e alla mia ragazza per avermi sopportato in questi cinque anni.

SOMMARIO

Nella tesi viene presentato il progetto e lo sviluppo di un ambiente software di supporto al progettista della parte elettrica di impianti di automazione industriale. L'ambiente software utilizza tecnologia Object-Oriented, Relazionale ed Intelligenza Artificiale e fornisce uno strumento di interfaccia fra visione object-oriented e relazionale delle informazioni.

Indice

Capitolo 1

Introduzione

Il presente lavoro di tesi nasce da una pre-esistente collaborazione con la ditta A.S.T. System Automation di Soliera (MO).
Tale ditta aveva identificato un preciso obiettivo da raggiungere: la realizzazione di un software che sia di supporto e di aiuto alla stesura della documentazione tecnica e commerciale relativa alla progettazione della parte elettrica di un impianto di automazione industriale.

Il progettista di impianti elettrici industriali deve fornire ai propri clienti:

- lo **schema elettrico** dell'impianto
- la lista dei materiali utilizzati
- la collocazione dei materiali nella fabbrica dove verra' eseguita materialmente l'automazione (**schema topografico**)

Attualmente, chi opera nel settore della automazione industriale, trova numerosi pacchetti software che assistono il progettista nell'attivita' di disegno degli schemi elettrici, cioe' sistemi CAD (Computer Aided Design) disponibili sul mercato.

Piu' difficile risulta la ricerca di strumenti in grado di svolgere sia le attivita' di progetto citate che quelle di catalogazione e reperimento dei materiali presenti in un impianto elettrico.

Da questi problemi, e' nata l'idea di progettare e realizzare un ambiente software di ausilio al progettista di impianti elettrici in grado di memorizzare e gestire tutte le informazioni di un impianto di automazione industriale,

partendo dallo schema elettrico (che risulta essere sempre il documento principale a cui fare riferimento) fino ad arrivare a preventivi e consuntivi sui costi di progetto.

L'attivita' di tesi ha quindi riguardato lo studio di fattibilita', il progetto e la realizzazione di un'ambiente software denominato "**Elet-Designer**".
Alcune componenti di Elet-Designer sono state sviluppate con Microsoft Visual C++ in ambiente Microsoft Windows NT. L'ambiente software e' basato su un database contenente tutti i dati relativi alla parte elettrica di un impianto di automazione industriale.
Il suddetto database e' stato progettato utilizzando **ODB-Tools**, uno strumento sviluppato in ambiente UNIX presso il Dipartimento di Scienze dell'Ingegneria dell'Universita' di Modena.
Inoltre sono state studiate e sviluppate estensioni di ODB-Tools per migliorare l'efficacia nell'aiuto al progettista di basi di dati complesse.

Il lavoro che segue tende a dimostrare come ODB-Tools, nonostante presenti alcuni aspetti carenti, sia un valido strumento di ausilio alla progettazione. ODB-Tools se ben utilizzato e potenziato con le estensioni sviluppate nella presente tesi assiste efficacemente il progettista di basi di dati.
Questo aiuto si e' evidenziato soprattutto nella realizzazione di Elet-Designer.

La tesi e' articolato in cinque parti, che corrispondono alle fasi del lavoro:

- analisi dei requisiti del dominio Elet
- ampliamento di ODB-Tools per rispondere alle nuove esigenze introdotte dal dominio Elet
- progettazione del database Elet con ODB-Tools
- sviluppo di un prototipo di interfaccia tra il database e il progettista di impianti elettrici, denominato "**Prelet**"
- sviluppo di uno strato software, denominato "**Object-World**", che permette a Prelet di usare le potenzialita' di un OODBMS (Object Oriented Database Management System) pur avendo a disposizione un RDBMS (Relational Database Management System).
  In particolare la scelta di realizzare Object-World nasce dalla piu' consolidata e diffusa tecnologia relazionale rispetto alla piu' recente tecnologia ad oggetti.

Nel capitolo 2 viene descritto il dominio applicativo in modo informale, quindi si presenta il ruolo dell'impianto elettrico in una realta' di automazione industriale e la metodologia di progettazione di tale impianto, adottata all'interno della ditta A.S.T. System Automation.

Nel capitolo 3 viene introdotto l'ambiente software Elet-Designer, la sua architettura e le sue componenti.

Nel capitolo 4 viene formalizzata l'analisi del componente Prelet e la descrizione dello schema del database mediante ODB-Tools.

Nel capitolo 5 viene descritto in modo approfondito il componente Object-World.

Nel capitolo 6 viene riassunta brevemente l'architettura di ODB-Tools e il suo formalismo OCDL, pre-esistenti al presente lavoro di tesi. Si riporta inoltre un'analisi critica dello strumento.

I capitoli 7 e 8 sono dedicati rispettivamente alla progettazione e alla implementazione di estensioni di ODB-Tools atte a superare i limiti precedentemente indicati.

Infine il capitolo 9 riassume il lavoro svolto e descrive lo schema del database progettato mediante ODB-Tools, con particolare attenzione alle regole.

Capitolo 2

Il progetto Elet-Designer

2.1 Analisi dei requisiti

2.1.1 Cosa significa "parte elettrica di un impianto di automazione industriale"

Nella realta' industriale odierna per svolgere i diversi compiti della produzione si adotta il lavoro combinato di diversi macchinari, ad esempio caricatori e bracci automatici. Questi sono degli apparati sostanzialmente meccanici, i quali vengono azionati e pilotati da apparecchi elettrici ed elettronici. Quando si progetta un processo produttivo automatico e' necessario sincronizzare tutte le operazioni eseguite dalle macchine in modo da svolgere correttamente la produzione.
Oggigiorno si opera in questo modo:

- si concentra il controllo del processo automatico in uno o piu' PLC (Programmable Logic Controller)
- il PLC viene collegato attraverso cavi elettrici a trasduttori
- i trasduttori sono di due tipi :
  - **sensori** - in grado di catturare informazioni dai macchinari e inviarli al PLC sotto forma di segnali elettrici
  - **attuatori** - in grado di prelevare il segnale elettrico del PLC e convertirlo in azioni da intraprendere sui macchinari

Affinche' il PLC sia in grado di colloquiare con il mondo reale (rappresentato dal processo produttivo) e dirigere automaticamente tutti i macchinari, e' ne-

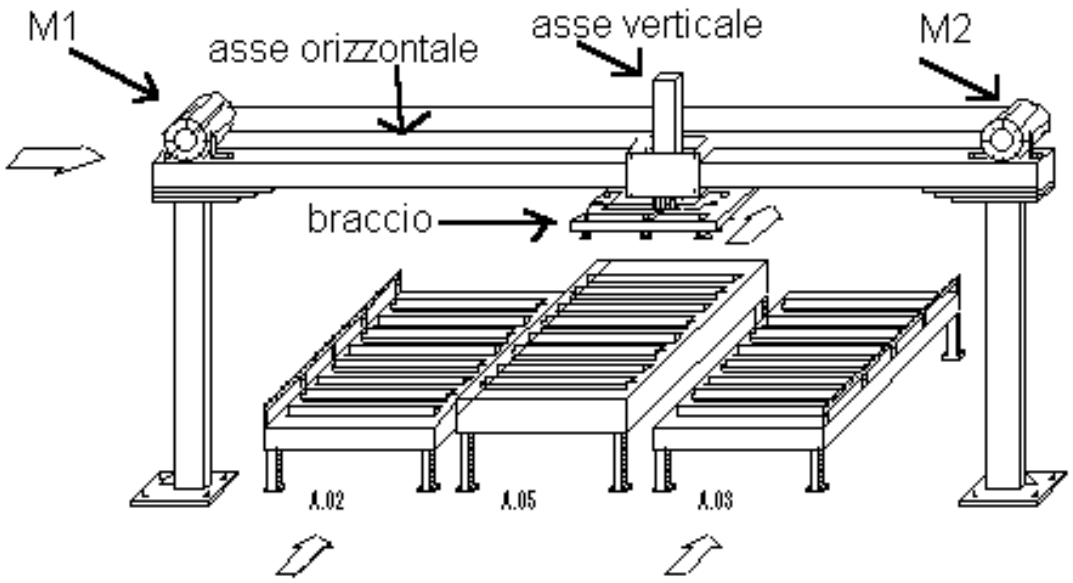

Figura 2.1: Esempio di braccio meccanico

cessario progettare un impianto elettrico piuttosto complesso. Tale impianto presenta diversi aspetti:

- problemi logistici di collocazione dei sensori e degli attuatori nella fabbrica
- problemi di cablaggio dei cavi elettrici
- problemi di sicurezza dell'impianto elettrico, da considerare con estrema importanza
- problemi di manutenzione dell'impianto elettrico

Il progettista deve riuscire a far funzionare in modo automatico i macchinari e deve assicurare una risposta adeguata a tutti i suddetti problemi.

In figura 2.1 viene riportato l'esempio di un macchinario industriale.
Questa macchina e' composta da un braccio che solleva dei pannelli di legno.
Il braccio puo' fare spostamenti orizzontali e verticali (questo avviene attraverso i due assi indicati in figura).
Per poter spostare il braccio sono necessari:

- due attuatori, in questo caso si tratta di **motori elettrici**: 'M1' per lo spostamento orizzontale e 'M2' per lo spostamento verticale.

- due sensori di posizione, detti **encoders**, per poter controllare in che posizione si trova il braccio in ogni istante.

2.1.2 Come si progetta la parte elettrica di un impianto di automazione industriale

Da una attenta analisi di anni di esperienza lavorativa nel settore automazione industriale si sono individuate le principali fasi di un progetto di un impianto elettrico:

1. Il cliente invia alla ditta il **layout** dell'impianto, ossia un disegno che rappresenta la mappa della fabbrica che si deve automatizzare, raffigurante tutti i macchinari (parte meccanica).
   Un esempio di layout e' il disegno del braccio meccanico riportato sopra.

2. Il cliente invia i dati tecnici generali dell'impianto, come ad esempio la temperatura in fabbrica, le linee elettriche gia' presenti ecc..

3. Il cliente spiega al progettista, in modo informale, le modalita' con cui deve essere comandato l'impianto e in che modo si devono muovere tutti i macchinari.

4. il progettista inserisce sul layout del cliente tutti gli attuatori ed i sensori necessari, cercando tra i vari cataloghi delle case costruttrici di apparecchi elettrici quelli che piu' si adattano alla situazione.
   D'ora in avanti chiameremo sensori e attuatori (che comprendono i motori) con il termine di **elementi primari**.
   Questa fase viene detta in gergo "disposizione degli elementi in **campo**", ossia direttamente nella fabbrica.

5. il progettista decide, per ogni elemento primario disposto sul layout, quali componenti, elettrici ed elettronici, sono necessari al suo comando e alla sua protezione elettrica.
   Questo e' un aspetto importantissimo, ogni elemento primario per poter operare ha bisogno di componenti elettrici di azionamento e di protezione (ad esempio i fusibili proteggono il motore elettrico). Tutti questi componenti sono logicamente correlati agli elementi primari e saranno quindi chiamati in seguito **sottoelementi**.

Bisogna osservare che i sottoelementi possono essere ricavati automaticamente, senza attendere il loro inserimento dal progettista, dalla conoscenza dello schema elettrico dell'elemento primario.
Questo aspetto sara' comunque approfondito successivamente.

6. il progettista organizza il cablaggio dei cavi e la disposizione di tutti i sottoelementi dentro a carpenterie che fungono da raccoglitori di componenti elettrici. Le carpenterie sono state denominate **elementi ausiliari**, in quanto non sono componenti elettrici, ma servono solo di ausilio alla parte elettrica.
   Gli elementi ausiliari sono essenzialmente tre:

   - **armadi** (o quadri elettrici) - raccolgono tutti i componenti elettrici, sono i luoghi dove si interviene con operazioni di manutenzione
   - **cassette di derivazione** - hanno il compito di raccogliere tutti i cavi che arrivano dai trasduttori e mandarli attraverso un unica canalina agli armadi (quindi hanno solo la funzione di tenere ordinati i cavi)
   - **consolle e pulsantiere locali** - raccolgono gli elementi di interfaccia tra l'impianto e gli esseri umani. I pulsanti, i selettori, gli indicatori analogici ed i display che servono all'essere umano per poter intervenire sul processo produttivo sono raccolti dentro a queste consolle. Tutti questi elementi sono chiamati **elementi di colloquio con l'uomo**.

   Il progettista deve consegnare al cliente lo schema di collocazione di tutti i sottoelementi nei vari elementi ausiliari, in modo che i cablatori, che costruiranno fisicamente i quadri elettrici, sappiano stabilire con esattezza dove allocare la componentistica elettrica.

7. Il progettista colloca il PLC in un particolare armadio detto **armadio di controllo**. In seguito egli stabilisce il numero e la tipologia di schede da inserire nel PLC e fissa tutti i collegamenti tra il PLC ed i trasduttori disposti nella fabbrica.

8. Il progettista, in collaborazione con un disegnatore CAD, redige lo schema topografico, ossia il layout del cliente "riempito" con gli elementi primari e i disegni delle carpenterie con tutti gli elementi in esse contenuti.

9. Il progettista compila la distinta materiali, il preventivo o il consuntivo sui costi di acquisto dei materiali.

10. Il progettista e il disegnatore CAD redigono lo schema elettrico

11. Con la fine della attivita' del progettista della parte elettrica, il lavoro continua nelle mani del programmatore del PLC, il quale deve disporre di tutte le informazioni topografiche ed elettriche della fabbrica.

Queste sono, a grandi linee, le fasi del lavoro. C'e' da osservare che questo processo non e' continuo e lineare, ma soggetto a modifiche, dovute a nuove esigenze del cliente oppure a nuove soluzioni del progettista.

Capitolo 3

Progetto dell'ambiente software

Il software che si intende realizzare, oggetto della presente tesi, dovra' aiutare il progettista in tutte le fasi elencate nel capitolo 2; dallo schema elettrico alla gestione commerciale.

Nel sistema informatico "legacy", cioe' usato nella ditta A.S.T. System Automation prima del presente progetto, tutte le informazioni utilizzate dal progettista erano contenute nello schema elettrico e nello schema topografico.
La prima attivita' e' stata quindi quella di analizzare questi schemi per ricavarne la conoscenza "intensionale", cioe' le strutture generali di organizzazione delle informazioni.

Lo schema intensionale cosi' ottenuto e' stato poi tradotto in uno schema di database; questo database ha una organizzazione dei dati molto diversa rispetto allo schema elettrico.
Infatti, la difficolta' che si incontra nel cercare di estrarre informazioni dallo schema elettrico sta nella sua organizzazione, mirata all'espressione grafica degli elementi e non al loro contenuto informativo.

Il progettista dovra' interagire con il software, e quindi con il suddetto database, durante tutta la progettazione, seguendo una ben precisa metodologia di lavoro stabilita in fase di analisi.
Il software guidera' il progettista nella successione delle fasi del lavoro, ma gli lascera' anche la necessaria flessibilita' di modifica del progetto.
Durante la progettazione il database verra' continuamente arricchito di nuove informazioni, cosi' facendo al termine di questo "processo di arricchimento" si avranno tutti i dati relativi all'impianto da realizzare.

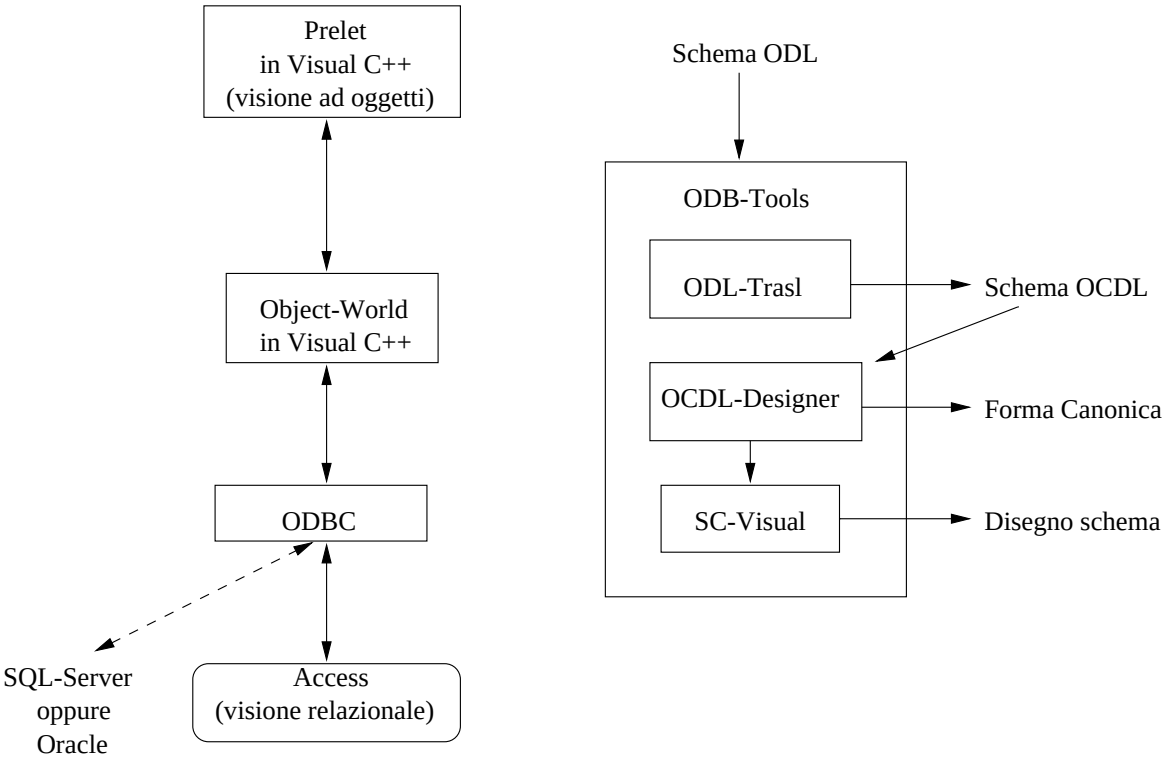

Figura 3.1: Schema generale di Elet-Designer

Tali dati rappresentano le informazioni necessarie e sufficienti alla stesura dello schema topografico, lo schema elettrico, la distinta materiali e il preventivo di spesa.

3.1 Architettura dell'ambiente software

In figura 3.1 si riporta uno schema che rappresenta in modo sintetico l'architettura di Elet-Designer e le relazioni fra le componenti sviluppate.

Osservando tale figura si puo' notare sulla sinistra la parte applicativa, sviluppata in ambito A.S.T. System Automation. Sulla destra della figura e' presente ODB-Tools, il quale permette di progettare lo schema della base di dati.

Le componenti di Elet-Designer sono:

1. **Prelet**
   Applicazione, realizzata con Microsoft Visual C++, con la quale il progettista di impianti elettrici interagisce con il database.
   Prelet mostra sul video i layout della fabbrica e permette al progettista di collocare graficamente gli elementi primari su tali layout. Durante tale processo si popola il database con gli elementi primari e con i sottoelementi, i quali vengono ricavati automaticamente dal programma.

2. **Object-World**
   Driver software, scritto in Visual C++, che permette ad una applicazione object oriented di rendere persistenti i propri oggetti in un database relazionale.
   Object-World si interfaccia al DBMS relazionale mediante ODBC[1] (Open DataBase Connectivity), esso attualmente utilizza Microsoft Access, ma in futuro potra' servirsi di altri DBMS aventi maggiori prestazioni.

3. **ODB-Tools**
   Sistema per l'acquisizione e la verifica di consistenza di schemi e per l'ottimizzazione semantica di interrogazioni nelle Basi di Dati Orientate agli Oggetti (OODB).
   ODB-Tools, dopo essere stato opportunamente esteso, e' stato impiegato per progettare lo schema del database.

3.1.1 Prelet

Cerchiamo di formalizzare alcuni aspetti del prototipo software "Prelet" utilizzando la metodologia OMT.

In figura 3.2 si illustra solamente l'organizzazione di alto livello del software, utilizzando lo schema DFD della metodologia OMT.

Nello schema si mettono in evidenza i seguenti agenti esterni:

- Il progettista, che fornisce tutto l'input necessario al sistema software (ossia il layout del cliente, i dati generali di impianto e e gli elementi)

- Il manutentore dei dati storici, e' una figura fittizia creata per evidenziare il fatto che si mantiene un database "storico" con tutti i materiali,

[1] per ulteriori informazioni riguardanti ODBC si veda appendice B

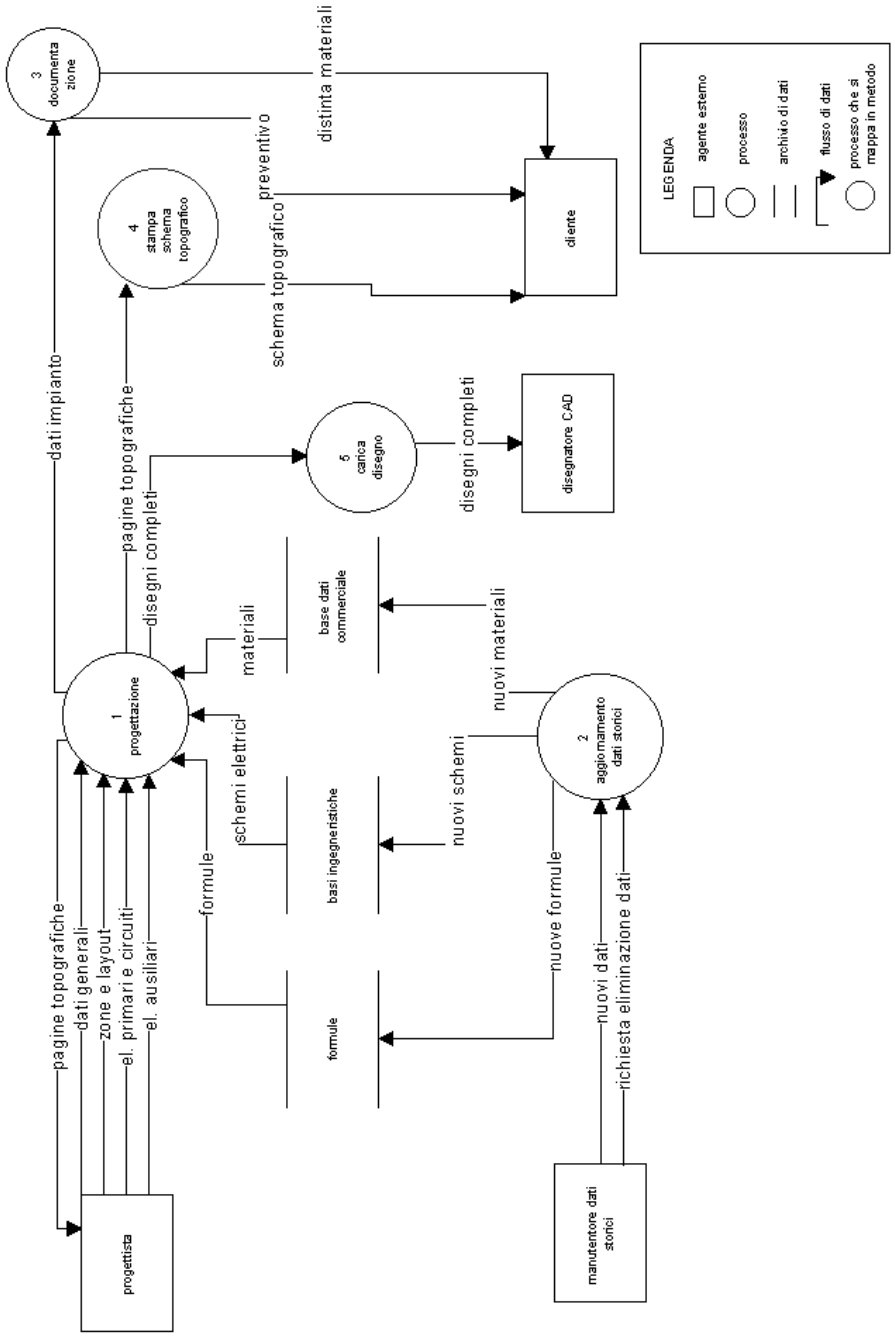

Figura 3.2: DFD di primo livello

gli schemi elettrici usati con piu' frequenza; dal quale il progettista (e quindi il software) puo' ricavare le informazioni necessarie a produrre l'impianto elettrico.

Nello schema si mettono in evidenza i seguenti dati storici:

- **Materiali**, sono tutti i componenti elettrici presenti a catalogo. Il software puo' interrogare questo archivio per cercare i componenti piu' adatti da utilizzare.
- **Basi ingegneristiche**, con questo termine ci si riferisce agli schemi elettrici degli elementi primari. Bisogna precisare che ogni volta che si progetta un nuovo impianto non si disegnano ex-novo gli schemi elettrici, ma si cerca di recuperare schemi gia' disegnati in precedenti occasioni. Quindi si e' deciso di conservare in un database storico tutti i modelli dei vari circuiti elettrici che si utilizzano, in modo che facilmente il progettista li possa riutilizzare.
- **Formule**, normalmente le grandezze elettriche dei sottoelementi sono legate alle medesime grandezze degli elementi primari a cui sono associati. Il progettista deve scegliere ogni sottoelemento in base alla corrente, la tensione o la potenza dell'elemento primario, applicando determinate regole derivanti dalla elettrotecnica.

Il centro dello schema e' rappresentato dal processo 1 denominato "progettazione". All'interno di esso e' contenuto il database centrale, il quale raccoglie tutte le informazioni di una singola istanza di impianto elettrico. Questo "database centrale" non e' da confondere con il database storico, il quale raccoglie informazioni legate alla esperienza della ditta.
E' evidente come dal database centrale si ricavano la documentazione e lo schema elettrico da fornire al cliente.

3.1.2 Object-World

La ditta A.S.T. System Automation progetta impianti industriali di notevoli dimensioni. La quantita' di informazioni che Prelet deve gestire e' ragguardevole. Microsoft Visual C++ non offre validi strumenti per rendere persistenti gli oggetti che vengono istanziati durante l'esecuzione dei programmi. Di conseguenza si e' reso necessario memorizzare gli oggetti in un database esterno. In mancanza di efficienti sistemi orientati agli oggetti, si e' scelto di affidarsi ad un DBMS relazionale.

Per memorizzare gli oggetti del Visual C++ all'interno di un DBMS relazionale, senza perdere la loro struttura, si e' dovuto creare uno strato software (denominato **Object-World**) di interfaccia tra il mondo object-oriented ed il mondo relazionale.

Questo strato software permette di rendere persistenti gli oggetti in un qualsiasi DBMS relazionale che supporti **Microsoft ODBC** (Open Database Connectivity). Per la realizzazione del prototipo e' stato scelto Microsoft Access in quanto gia' presente nella ditta.

Nelle applicazioni orientate agli oggetti le informazioni sono organizzate in collezioni di oggetti contenenti i dati e le operazioni. Nei database relazionali le informazioni sono memorizzate in tabelle. Object-World permette di conservare gli oggetti in tabelle senza enormi sforzi per il programmatore che lo utilizza.

3.1.3 ODB-Tools

Per la fase di progettazione di un database ad oggetti ODB-Tools puo' risultare molto utile. In particolare il progettista puo' avvalersi:

- di un linguaggio standard, ODL-ODMG 93, per la definizione della struttura dei database ad oggetti, fornito da ODL-Trasl
- dell'algoritmo di sussunzione che aiuta a stabilire l'esatta tassonomia delle classi
- del linguaggio ODL-esteso presente in ODB-Tools. Nel caso di database in cui esiste una forte presenza di vincoli sugli attributi, tale linguaggio permette di esprimere gia' nella parte intensionale la presenza di regole che rappresentano i vincoli.
- di strumenti che permettono di ricavare eventuali incoerenze nello schema del database (OCDL-Designer)
- di ODBQO per ottimizzare le numerose query del database

Alcune componenti di ODB-Tools sono state estese. In particolare le estensioni riguardano:

- l'introduzione di metodi nello schema delle basi di dati.

- l'introduzione di metodi nei predicati dei vincoli di integrita' al fine di arricchire le possibilita' espressive offerte dal linguaggio ODL-esteso.

Capitolo 4

Il componente Prelet

4.1 DFD di Prelet

Nel paragrafo 3.1.1 e' riportato solamente il diagramma DFD di primo livello. Nel presente capitolo si formalizza il prototipo software "Prelet" illustrando i vari livelli di DFD[1] e la struttura del database ottenuta utilizzando ODB-Tools.

In figura 4.1 e nelle successive figure si illustra l'organizzazione del software. Si allega, inoltre, per ogni livello di DFD, il dizionario dei dati, suddiviso per categoria di voce.

[1]per ulteriori informazioni riguardanti DFD si veda appendice E.1

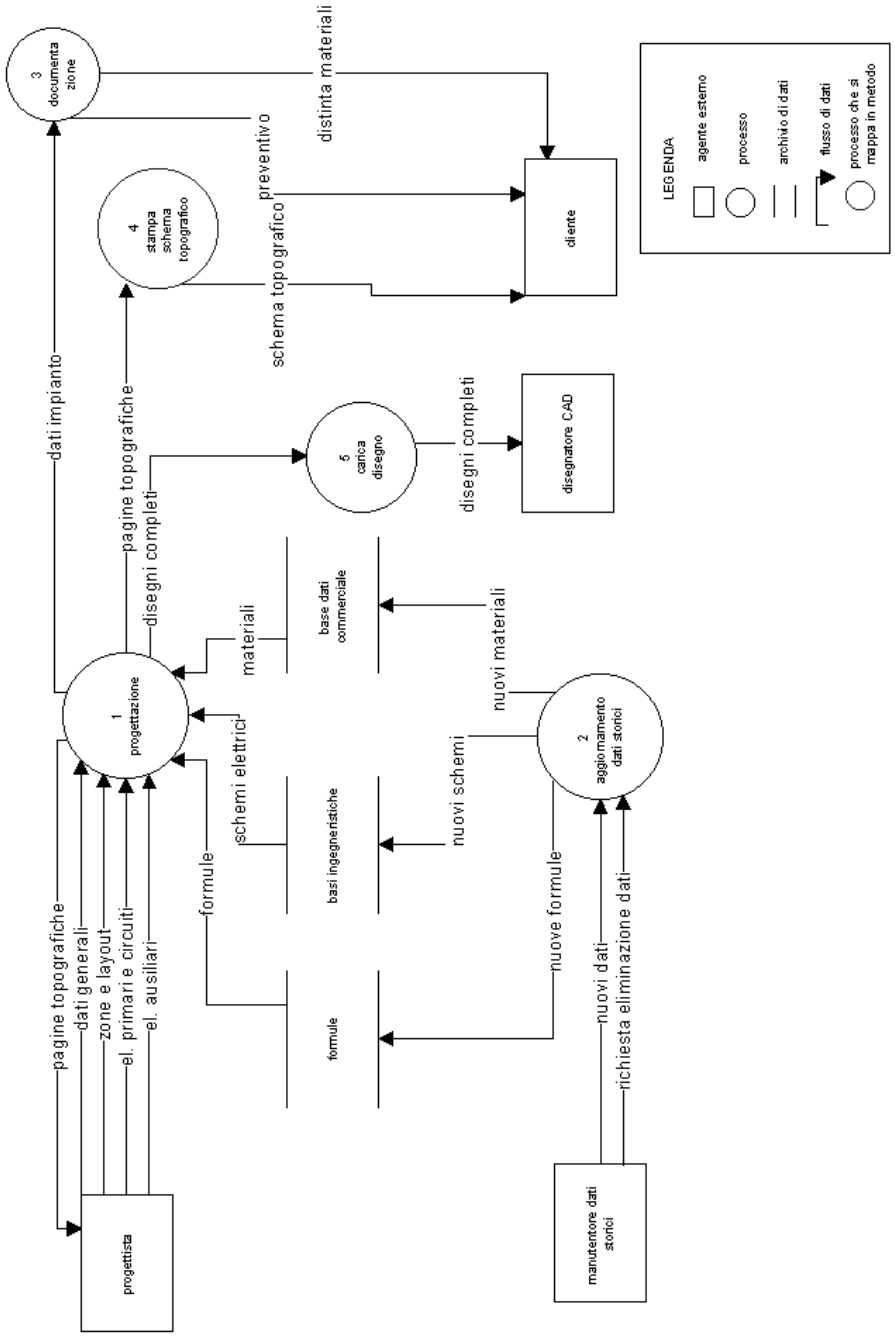

Figura 4.1: DFD di primo livello

AGENTI ESTERNI	DFD Generale
Cliente	Destinatario di tutta la documentazione
Disegnatore CAD	Persona addetta al disegno dello schema elettrico con uno strumento CAD
Manutentore dati storici	Responsabile del mantenimento del database "storico" con tutti i materiali e gli schemi elettrici
Progettista	Fornisce tutto l'input necessario al sistema software

FLUSSI DI DATI	DFD Generale
Circuiti	Sono i dati dei circuiti di alimentazione presenti in ogni zona; essi vengono trattati come elementi primari, ma non presentano simboli sul layout
Dati generali	Tutti i dati generali al progetto
Dati impianto	Sono tutti i dati alfanumerici dell'impianto
Disegni completi	Sono gli schemi elettrici degli elementi con l'indicazione delle caratteristiche elettriche di ogni componente presente al loro interno.
Distinta materiali	Elenco di tutti i materiali utilizzati
El. Ausiliari	Sono i dati relativi alle carpenterie
El. Primari	Sono i dati relativi agli elementi primari
Layout	Sono i disegni della mappa dell'impianto. Generalmente sono necessarie molte pagine di layout, quindi ogni zona possiede diversi layout.
Materiali	Tutti i componenti elettrici presenti a catalogo
Pagine topografiche	Sono i layout completi dei simboli degli elementi disposti su di essi
Preventivo	Distinta dei materiali con i prezzi di ogni materiale e il costo totale preventivato in sede di progettazione
Rich. eliminazione dati	Richiesta di cancellare un materiale o uno schema elettrico
Schema topografico	Elenco di tutte le pagine topografiche arricchite dai simboli degli elementi primari disposti su di essi
Schemi elettrici	Disegni degli schemi elettrici degli elementi primari, servono per derivarne i sottoelementi, i morsetti, gli IO_PLC e i fili di collegamento
Zone	Zone in cui viene suddiviso l'impianto, esiste sempre almeno una zona denominata "campo" e una zona per ogni elemento ausiliario

PROCESSI	DFD Generale
Aggiornamento dati storici	Processo di aggiornamento del database storico formato dai materiali commerciali (da catalogo)
Carica disegno	Processo che permette di caricare un disegno completo e inserirlo nel programma CAD che si sta utilizzando
Documentazione	Processo che si occupa della stesura della documentazione alfanumerica
Progettazione	Processo principale che gestisce tutti i dati dell'impianto che si sta realizzando; gestisce la grafica e il database centrale
Stampa schema topografico	Processo che stampa tutti i layout, cioe' lo schema topografico completo

ARCHIVI DATI	DFD Generale
Base dati commerciale	Tutti i materiali presenti nei cataloghi
Basi ingegneristiche	Schemi elettrici degli elementi primari. Si conservano nel database storico tutti i modelli dei circuiti elettrici che si utilizzano, in modo che facilmente il progettista li possa riutilizzare
Formule	Regole di elettrotecnica utili al progettista per dimensionare i componenti utilizzati nell'impianto

In figura 4.2 si riporta il processo 1 "progettazione". Di seguito il dizionario dei dati, nel quale si inseriscono solo i nuovi nomi, introdotti con l'esplosione di livello di DFD. Si rimanda ai dizionari dei dati precedenti la spiegazione di nomi gia' introdotti.

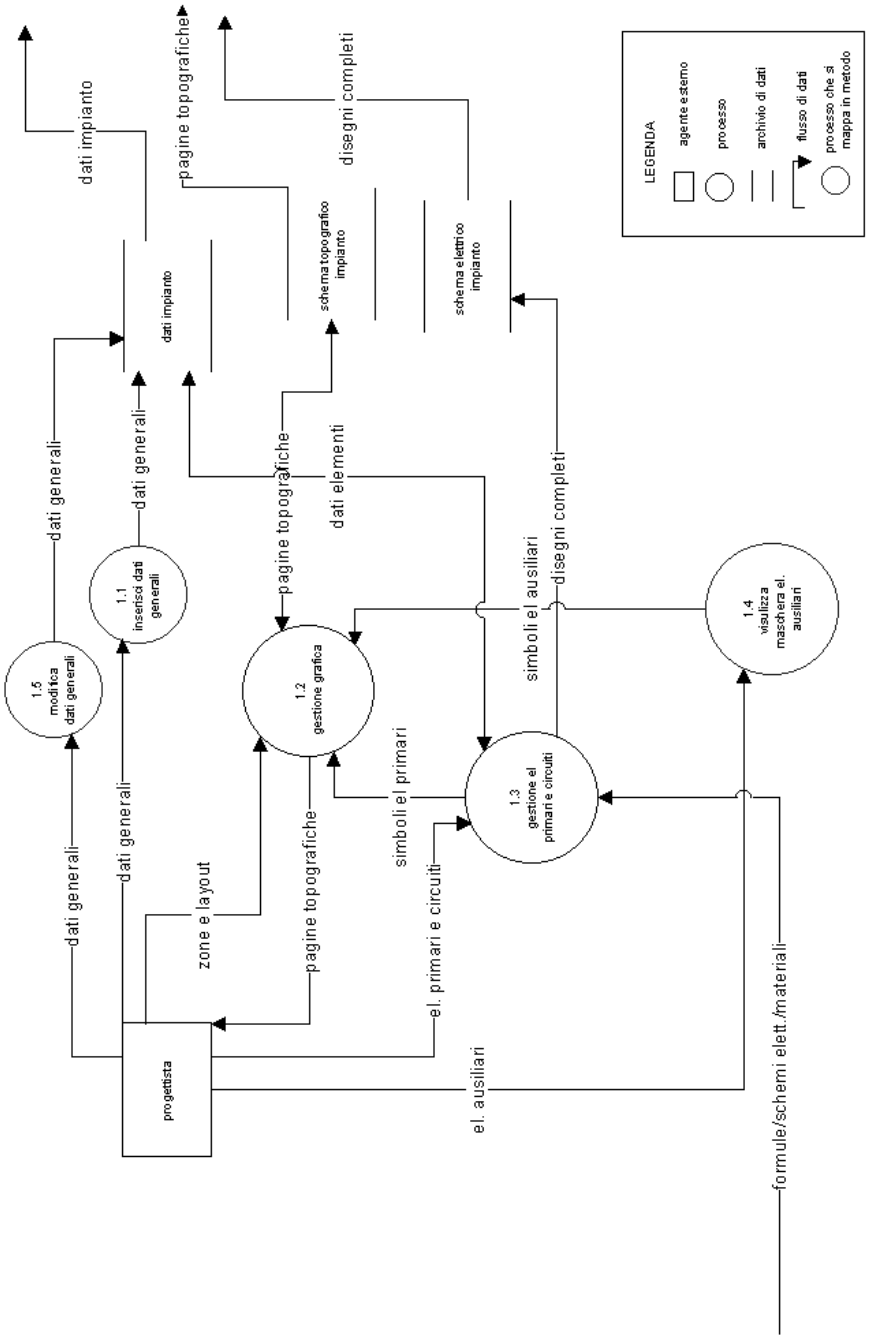

Figura 4.2: DFD del processo 1 : “progettazione”

FLUSSI DI DATI	processo "progettazione"
Dati elementi	Sono i dati dei vari elementi che dispongo sui layout. Questi vengono tutti inseriti nel database centrale dei dati dell'impianto
Schemi elett./materiali	Ho raggruppato in un unico flusso di dati le due frecce presenti al livello superiore di DFD
Simboli el. Ausiliari	Simboli da rappresentare sul layout relativi ad elementi ausiliari
Simboli el. Primari	Simboli da rappresentare sul layout relativi ad elementi primari
PROCESSI	**processo "progettazione"**
Gestione el. Primari e circuiti	Processo che si occupa di visualizzare una maschera per l'inserimento di tutti i dati relativi ad ogni elemento. Si occupa anche della derivazione dei sottoelementi, dei morsetti, degli IO_PLC e dei fili
Gestione grafica	Processo che si occupa della visualizzazione dei layout a video, dell'inserimento dei simboli degli elementi e della gestione delle zone
Inserisci dati generali	Processo che visualizza una maschera a video e chiede l'inserimento dei dati generali dell'impianto
Modifica dati generali	Processo che visualizza una maschera a video e chiede la modifica dei dati generali dell'impianto.
Visualizza maschera el. Ausiliari	Processo che si occupa di visualizzare una maschera per inserire i dati relativi agli el. Ausiliari
ARCHIVI DATI	**processo "progettazione"**
Dati impianto	Questo e' il database centrale relativo alla istanza di progetto che si sta realizzando
Schema elettrico impianto	Gli schemi elettrici completi sono raggruppati in questo archivio. Il disegnatore CAD potra' usare questo archivio per redigere con estrema facilita' lo schema elettrico dell'impianto
Schema topografico impianto	I layout sono raccolti in questo archivio. Bastera' accedere a questo archivio per ottenere tutto lo schema topografico

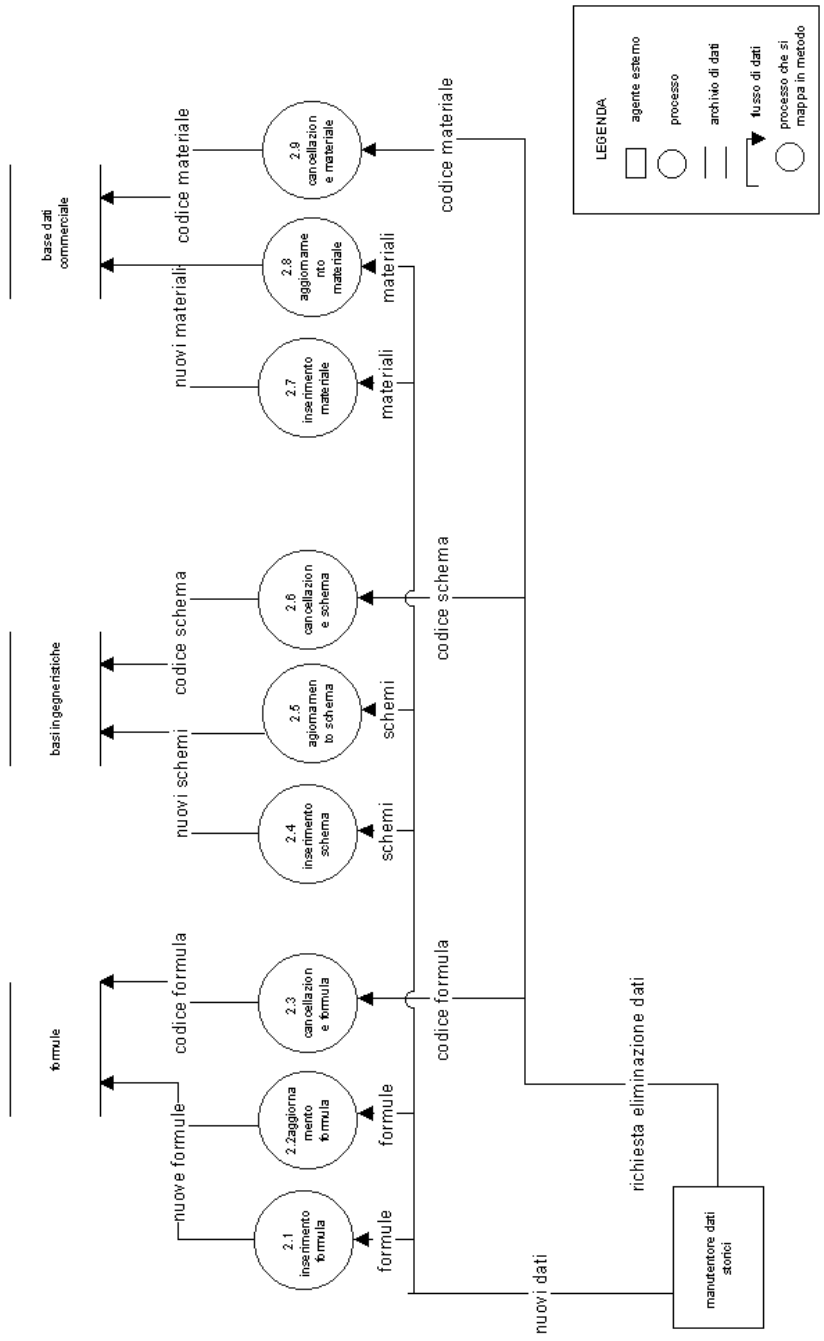

Figura 4.3: DFD del processo 2 : "aggiornamento dati storici"

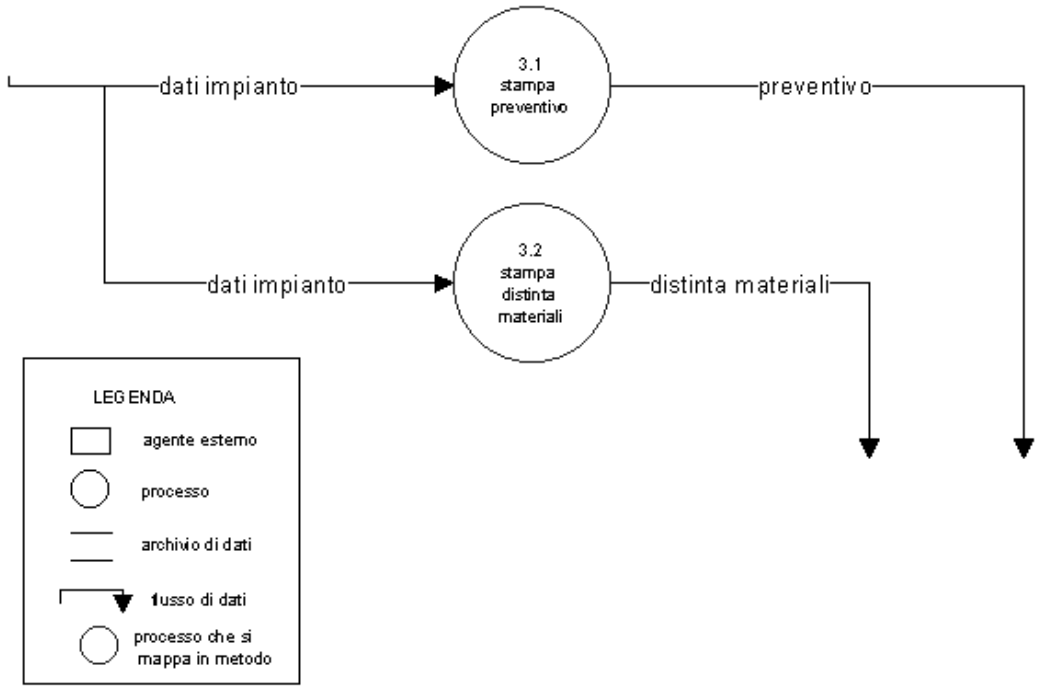

Figura 4.4: DFD del processo 3 : "documentazione".

FLUSSI DI DATI	processi 2 e 3
Codice materiale	Per la cancellazione di un materiale serve solo il suo codice
Codice schema	Per la cancellazione di una schema elettrico serve solo il suo codice
Dati impianto	Dati alfanumerici dell'impianto
El. Primari	Sono i dati relativi agli elementi primari
Nuovi materiali	Sono i nuovi materiali da inserire o aggiornare
PROCESSI	**processi 2 e 3**
inserimento materiale aggiornamento materiale cancellazione materiale	Processi che gestiscono il database storico dei materiali
inserimento schema aggiornamento schema cancellazione schema	Processi che gestiscono il database storico deglle basi ingegneristiche
Stampa distinta materiali	Processo che elenca i materiali, gli elementi ausiliari e gli elementi che essi contengono, i morsetti, gli IO_PLC e i fili di collegamento
Stampa preventivo	Processo che stampa una distinta materiale con l'indicazione dei prezzi e il totale dei costi

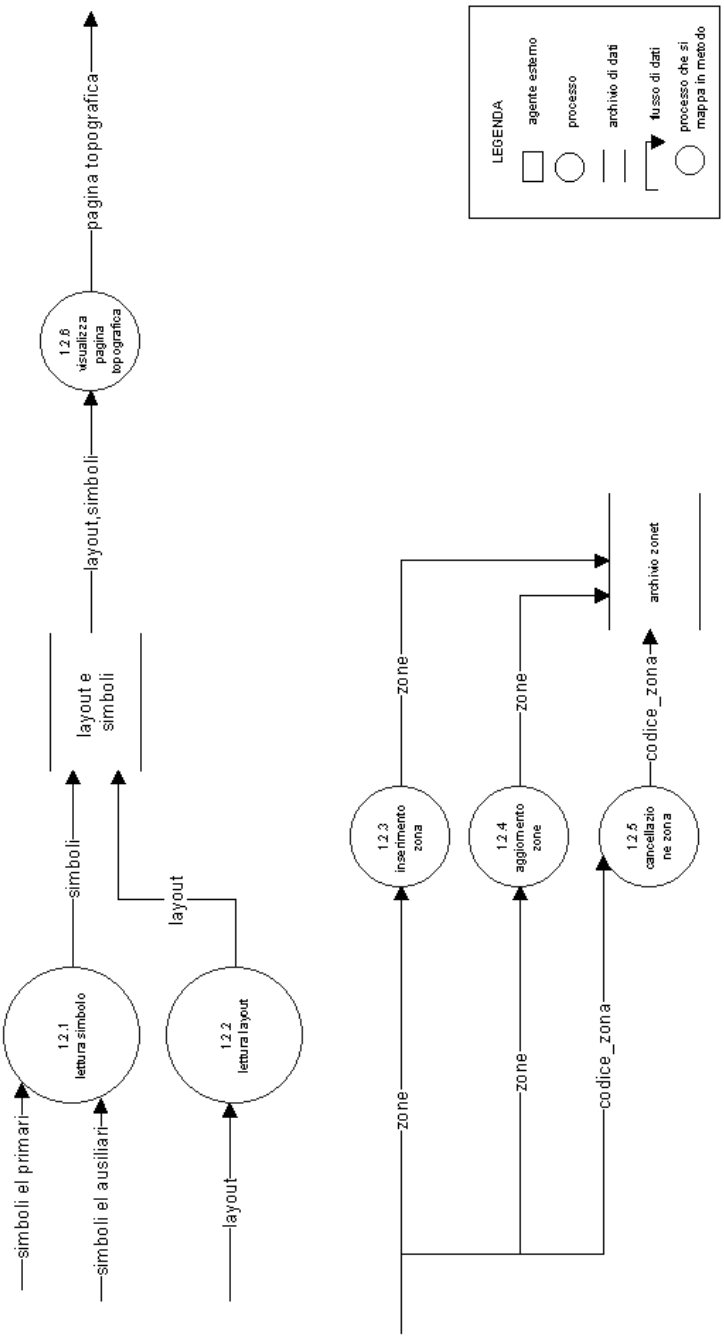

Figura 4.5: DFD del processo 1.2 : "gestione grafica"

FLUSSI DI DATI	processo 1.2
Codice zona	Per la cancellazione di una zona serve solamente il codice
Simboli	Sono i simboli da rappresentare sul layout relativi ad elementi primari
PROCESSI	**processo 1.2**
Lettura simbolo	Processo che legge un file grafico di un simbolo, interpretando correttamente il suo formato
Lettura Layout	Processo che legge un file grafico di un layout, interpretando correttamente il suo formato
Inserimento zona aggiornamento zona cancellazione zona	Processi che si occupano della gestione dell'archivio delle zone
Visualizza pagina topografica	Processo che si occupa di visualizzare a video un layout riempito di tutti i simboli
ARCHIVI DATI	**processo 1.2**
Archivio zone	Questo e' l'archivio che conserva l'indicazione delle zone dell'impianto.
Layout e simboli	I layout e i simboli sono singolarmente salvati in questo archivio. Questo serve per poter ridisegnare i singoli simboli per eventuali spostamenti sul grafico ecc..

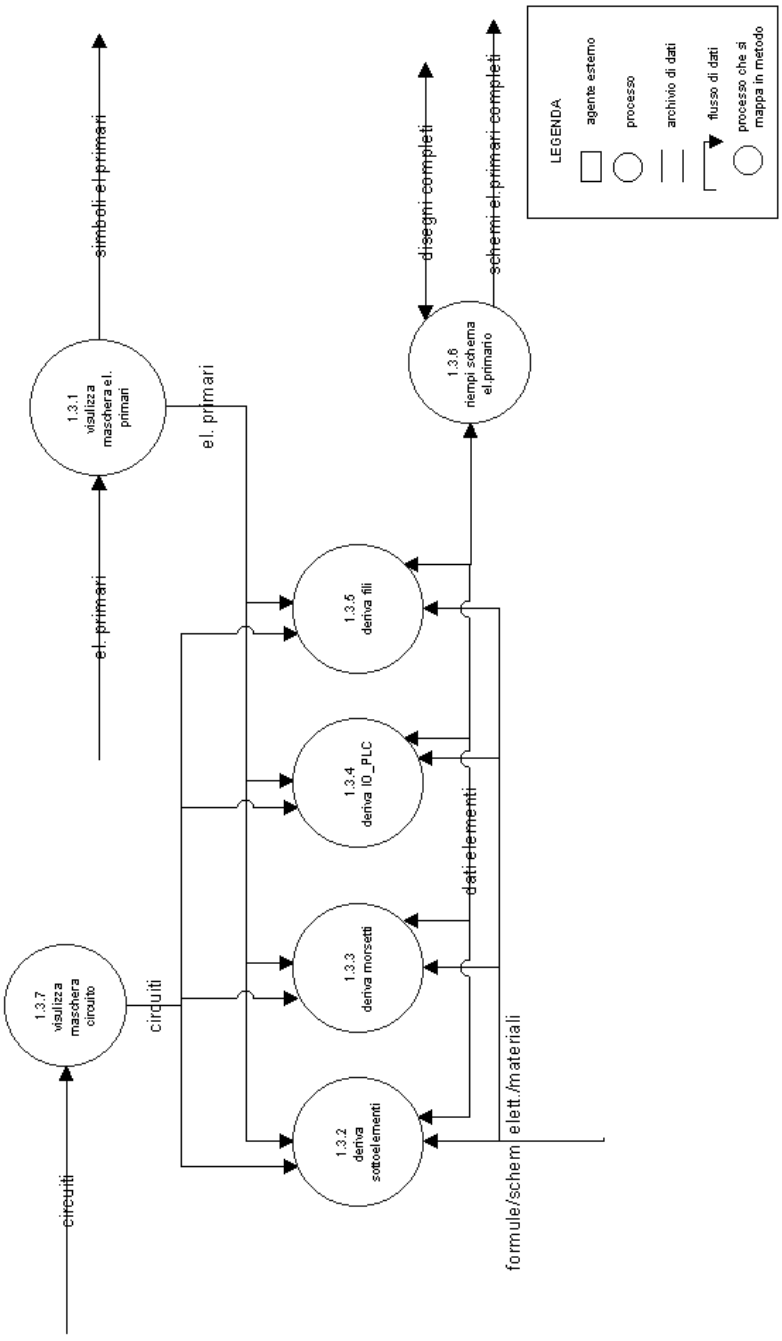

Figura 4.6: DFD del processo 1.3 : "gestione el. primari"

FLUSSI DI DATI	processo 1.3
Dati elementi e circuiti	Ho unito in un unico flusso i dati degli elementi e dei circuiti che servono per completare lo schema elettrico. Quindi mi riferisco in particolare a tutte le grandezza elettriche degli elementi e dei circuiti
PROCESSI	**processo 1.3**
Deriva fili	Processo che legge lo schema elettrico associato all'elemento primario e ne deriva i relativi fili di collegamento, inoltre associa in questo modo tutti i morsetti tra di loro in modo che sia possibile stabilire se ci sono componenti non collegati
Deriva IO_PLC	Processo che legge lo schema elettrico associato all'elemento primario e ne deriva i relativi IO_PLC
Deriva morsetti	Processo che legge lo schema elettrico associato all'elemento primario e ne deriva i relativi morsetti
Deriva sottoelementi	Processo che legge lo schema elettrico associato all'elemento primario e ne deriva i relativi sottoelementi, che saranno accodati agli altri per essere poi inseriti in qualche carpenteria
Riempi schema el. Primario	Processo che dato in input tutti i dati relativi ad un elemento primario e i suoi collegamenti con gli altri componenti e' in grado di prendere lo schema elettrico e riempirlo con tutte le caratteristiche elettriche, siglare i morsetti, numerare i fili ecc...
Visualizza maschera circuito	Processo che si occupa di visualizzare a video una maschera per l'inserimento dei dati relativi a un circuito di alimentazione
Visualizza maschera el. primario	Processo che visualizza una maschera per inserire un elemento primario

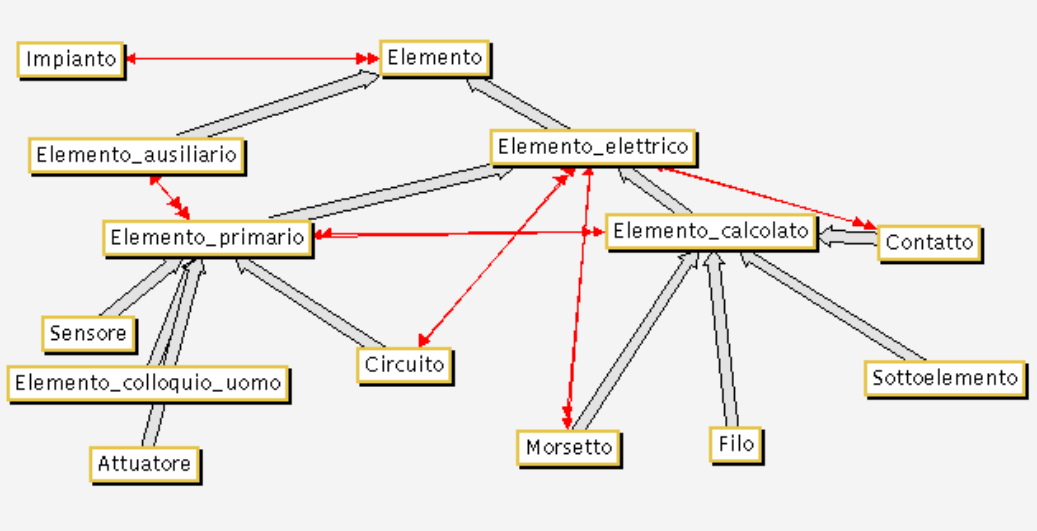

Figura 4.7: Schema del database

4.2 Schema concettuale del dominio Elet con regole in linguaggio naturale

E' stato scelto di realizzare lo schema concettuale con ODB-Tools, utilizzando le potenzialita' espressive di ODL-esteso.

Nel progettare lo schema concettuale del database, si e' tenuto conto:

- dell'input necessario ad un impianto elettrico
- delle interrogazioni che serviranno al progettista e al cliente finale
- delle informazioni utili alla documentazione tecnica e allo schema elettrico

Viene riportato in figura 4.2 la rappresentazione grafica dello schema del database ottenuto con l'applet java **scvisual** di ODB-Tools [2].
Si precisa che nello schema :

- sono indicati i nomi delle classi
- non sono indicati i nomi delle relazioni

[2]per ulteriori informazioni riguardanti l'applet java scvisual si veda [Cor97]

- non sono indicati gli attributi e i metodi, i quali sono espressi di seguito in forma tabellare

- Per i metodi si e' indicato il processo corrispondente nel diagramma DFD riportato nel paragrafo precedente

- non sono riportati i vincoli, i quali sono indicati di seguito alle tabelle relative alle classi. I vincoli sono espressi in un linguaggio naturale simile alla sintassi di ODL-esteso utilizzata all'interno di ODB_Tools. Ogni vincolo e' identificato da un nome alfanumerico, il quale permette di trovare la corrispondente rule definita nello schema ODL.
  I vincoli, o meglio le rule dello schema del database sono la formalizzazione delle "formule" introdotte nel diagramma DFD in figura 3.2.

IMPIANTO	
Codice_impianto	String
Descrizione_impianto	String
Luogo	String
Temperatura_minima	Integer
Temperatura_massima	Integer
Altitudine	Integer
Umidita_minima	Range{0,100}
Umidita_massima	Range{0,100}
Codice_cliente	String
Inserisci_dati_generali	- processo 1.1
Modifica_dati_generali	- processo 1.5
Stampa_schema_topografico	- processo 4
Stampa_preventivo	- processo 3.1
Stampa_distinta_materiali	- processo 3.2

```
(Temperatura_minima< Temperatura_massima)
(Umidita_minima< Umidita_massima)
```

N.B: manca un metodo per stampare lo schema elettrico, infatti il software restituisce solo parti di schema elettrico, che devono essere completate dal disegnatore CAD

ELEMENTO	
codice_elemento	String
casa_costruttrice	String
serie	String
modello	String
descrizione_elemento	String
protezione	String
catalogo_pagina	String
ulteriori_dati_tecnici	String
codice_principale_sigla	String
materiale_sigla	String
Stampa_dati_elemento	

```
(protezione[0]='I')AND(protezione[1]='P')
(codice_principale_sigla[0]='=')
(materiale_sigla[0]='-')
```

[3]

ELEMENTO_ELETTRICO
Stampa_dati_elemento

ELEMENTO_AUSILIARIO	
tipo_materiale	String
colore	String
larghezza	Float
altezza	Float
profondita	Float
Stampa_dati_elemento	

ELEMENTO_CALCOLATO	
tipo_elemento_calcolato	String
Stampa_dati_elemento	

ELEMENTO_PRIMARIO	
corrente_assorbita	Range{0,+∞}
Stampa_dati_elemento	
Deriva_sottoelementi	processo 1.3.2
Deriva_morsetti	processo 1.3.3
Deriva_fili	processo 1.3.5

[3] per capire questi vincoli si veda i paragrafi protezione e siglatura in appendice C

ATTUATORE	
potenza	Float
cosfi	range{0,1}
num_cicli_lavoro_per_ora	integer
num_ore_funz_al_giorno	range{0,24}
categoria_impiego	Integer
coefficiente_contemporaneita	Float
Stampa_dati_elemento	

```
(0<cosfi<1)

Att1 : if tipo alimentazione del circuito ='DC'
        then (Categoria_impiego in ('DC-20','DC-21', 'DC-22', 'DC-23'))
Att2 : if tipo alimentazione del circuito ='AC'
        then (Categoria_impiego in ('AC-20','AC-21', 'AC-22', 'AC-23'))
Att3 : (Corrente_assorbita=f(potenza,tensione,cosfi))
```

[4]

SENSORE	
uscita	String
Stampa_dati_elemento	

```
(uscita in ('digitale','analogica'))
```

ELEMENTO_COLLOQUIO_UOMO	
uscita	String
larghezza	Float
altezza	Float
profondita	Float
Stampa_dati_elemento	

```
(uscita in ('digitale','analogica'))
```

[4] per chiarimenti su questi vincoli si veda il paragrafo categorie di impiego in appendice C

SOTTOELEMENTO	
categoria_impiego	String
corrente_nominale	range{0,+∞}
potere_interruzione	range{0,+∞}
Stampa_sottoelemento	

```
Sott1: if tipo_elemento_calcolato='Q'
        then (categoria_impiego
        = categoria di impiego degli elementi primari da cui deriva)
Sott2 : if tipo_elemento_calcolato='Q'
        then (corrente_nominale
        = somma della corrente assorbita
         dagli elementi primari da cui deriva)
Sott3 : if tipo_elemento_calcolato='Q'
        then (potere_interruzione
        = f(categoria di impiego,corrente nominale)

Sott4 : if tipo_elemento_calcolato='F'
        then (corrente_nominale
        = somma della corrente assorbita
          dagli elementi primari da cui deriva
          maggiorata del 30 %)

Sott5 : if tipo_elemento_calcolato='K'
        then (categoria_impiego
        = categoria di impiego degli elementi primari da cui deriva)

Sott6 : if tipo_elemento_calcolato='K'
        then (corrente_nominale
        = somma della corrente assorbita
          dagli elementi primari da cui deriva)

Sott7 : if tipo_elemento_calcolato='K'
        then (potere_interruzione
        = f(categoria di impiego,corrente nominale)

Sott8 : if tipo_elemento_calcolato='T'
        then (corrente_nominale
        = somma della corrente assorbita
         dagli elementi primari da cui deriva)
```

5

FILO	
tipo_filo	String
corrente_impiego	range{0,+∞}
sezione	range{0,+∞}
Stampa_filo	

```
(tipo_filo in ('di fase','di neutro'))

Fil1 : (corrente_impiego=somma della corrente assorbita dagli elementi
                  primari da cui deriva)

Fil2 : if alimentato.tipo_circuito='potenza'
       then (sezione = corrente_impiego/2.5)

Fil3 : if alimentato.tipo_circuito='ausiliario'
       then (sezione = 1.5)
```

6

MORSETTO	
morsetto_sigla	String
corrente_impiego	range{0,+∞}
sezione	range{0,+∞}
Stampa_morsetto	

```
(morsetto_sigla[0]=':')

Mors1 : (corrente_impiego=somma della corrente assorbita dagli elementi
                  primari da cui deriva)

Mors2 : if alimentato.tipo_circuito='potenza'
        then (sezione = corrente_impiego/2.5)

Mors3 : if alimentato.tipo_circuito='ausiliario'
        then (sezione = 1.5)
```

7

[5] Questi vincoli sono stati tratti dalle normative CEI 17-11, DIN VDE 0660, parte 107/9.82

[6] Questi vincoli sono stati tratti da regole pratiche utilizzati comunemente dai progettisti di impianti elettrici

[7] per chiarimenti sul primo vincolo si veda il paragrafo siglatura in appendice C

CONTATTO	
tipo_contatto	String
Stampa_contatto	

```
(tipo_contatto in ('istantaneo NO','istantaneo NC',
                   'ritardato NO','ritardato NC'))
```

CIRCUITO	
tipo_linea	String
tipo_circuito	String
tipo_alimentazione	String
tensione	Float
frequenza	Float
Stampa_circuito	

```
(Tipo_linea in ('monofase',
                 'trifase con neutro','trifase senza neutro'))
(Tipo_circuito in ('potenza','ausiliario'))
(tipo_alimentazione in ('DC','AC'))
(Frequenza in (50,60,100,120,300,400))

Circ1 : if tipo_alimentazione='DC'
        then (Tensione in (6,12,24,36,48,60,96,110,220,
                    440,660,750,1200,1500,2400,3000))

Circ2 : if tipo_alimentazione='AC'
        then (Tensione in (6,12,24,48,60,110,125,220,
                     380,500,660,1000,3000,6000,10000))

Circ3 : (corrente_assorbita=funzione delle utenze che alimenta)
```

[8]

Di seguito vengono descritte tutte le classi rappresentate in figura 4.2, per ognuna di esse si spiega brevemente il significato dei suoi attributi.

- **Impianto** Un impianto e' formato da elementi, ossia componenti elettrici e non elettrici.
  Descrizione degli attributi:
  - Luogo - Collocazione dell'impianto

[8] queste informazioni sono state tratte dalle normative DIN IEC 38 del 5/87 per le tensioni, le frequenze sono state tratte dagli usi piu' frequenti

  - Temperatura_minima e Temperatura_massima - Range di temperatura in cui lavora l'impianto elettrico
  - Umidita_minima e Umidita_massima - Range di umidita relativa in cui lavora l'impianto elettrico[9]

- **elemento** Classe astratta che generalizza il concetto di elemento di un impianto elettrico. Un elemento puo' essere un elemento elettrico o un elemento non elettrico (chiamato elemento ausiliario).
  Descrizione degli attributi:

  - codice_elemento - Codice univoco per ogni elemento secondo una codifica interna alla ditta (da non confondere con la siglatura spiegata in appendice C)
  - casa_costruttrice, serie e modello - Casa costruttrice, serie e modello del componente
  - protezione - Codice indicativo delle protezioni che possiede l'elemento secondo una normativa comunitaria (si veda appendice C)
  - catalogo_pagina - Catalogo e pagina dove reperire ulteriori informazioni tecniche
  - Ulteriori_dati_tecnici - Campo libero dove poter inserire altre caratteristiche tecniche utili al progettista
  - codice_principale_sigla e materiale_sigla - Questi sono i due campi che contengono le due parti della sigla dell'elemento (vedi il paragrafo siglatura in appendice C)

- **elemento_elettrico** Un elemento elettrico puo' essere un elemento primario o un elemento calcolato. Un qualsiasi elemento elettrico, per definizione, contiene uno o piu' morsetti di collegamento.

  Gli attributi rappresentano caratteristiche elettriche dell'elemento.

- **elemento_ausiliario** Un elemento ausiliario puo' essere un armadio, una cassetta di derivazione oppure una consolle-pulsantiera locale. In realta' la suddivisione in 3 classi figlie non e' stata realizzata poiche' non ci sono attributi significativi per separare le 3 classi, quindi i diversi tipi di elementi ausiliari vengono tutti istanziati nella classe elemento_ausiliario. Sono presenti 3 relazioni:

[9] Questi dati influiscono sulla scelta dei materiali

  - contiene - gli elementi primari collegati mediante questa relazione sono effettivamente contenuti nell'elemento ausiliario (e di conseguenza sono contenuti in esso anche i suoi sottoelementi)
  - contiene_parti_di - in questo caso l'elemento primario referenziato non e' contenuto nell'elemento ausiliario, ma solo i suoi sottoelementi.
  - collegato - in questo caso ne' l'elemento primario referenziato e ne' i suoi sottoelementi sono contenuti nell'elemento ausiliario, pero' devono figurare dei morsetti di collegamento perche' dall'elemento ausiliario partono dei fili di collegamento verso essi.

  Descrizione degli attributi:

  - tipo_materiale e colore - proprieta' esterne della carpenteria
  - larghezza, altezza, profondita - dimensioni della carpenteria

- **elemento_primario** Questa e' la classe degli elementi primari disposti sull'impianto. Come si puo' notare essi si suddividono in attuatori, sensori e attuatori e in elementi di colloquio con l'uomo (gia' spiegati nel capitolo 2).

- **elemento_calcolato** Questa e' la classe degli elementi che vengono calcolati dal programma. Questo significa che essi non vengono dati in input dal progettista, bensi' sono derivati dalla lettura delle basi ingegneristiche associate agli elementi primari. Essi sono :

  - Sottoelementi
  - Fili
  - Morsetti
  - Contatti
  - I/O PLC

- **Circuito** Questa classe mette in evidenza i circuiti di alimentazione presenti nell'impianto. Anche i circuiti, come gli elementi primari hanno dei sottoelementi (es. un interruttore principale o un trasformatore).

- **Morsetto** Un Morsetto e' una terminazione di un elemento primario, e' importante conservare nel database i morsetti perche' essi vengono raggruppati insieme per costruire delle morsettiere.

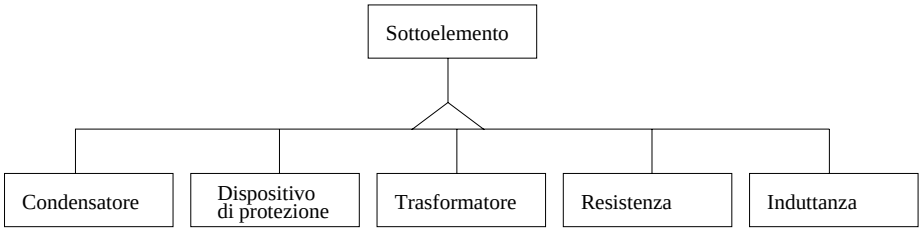

Figura 4.8: Esempio di gerarchia di sottoelementi

Per quanto concerne la descrizione degli attributi si puo' osservare che e' stato aggiunto un nuova campo relativo alla sigla dei morsetti. Infatti i morsetti presentano una ulteriore sigla che inizia con il carattere ":"[10].

- **Sottoelemento** Questa e' la classe di componenti elettrici che servono per comandare gli elementi primari (ad es. contattori, sezionatori, fusibili ecc..). Da notare che anche i sottoelementi hanno dei morsetti di collegamento. Anche questi sottoelementi potrebbero essere distinti in sottoclassi, un esempio e' riportato in figura 4.8.

  Descrizione degli attributi:

  - Tipo_uscita - specifica se l'uscita e' digitale o analogica
  - num_contatti_istantanei_NC, num_contatti_ritardati_NC - numero dei contatti da comandare (aprire e chiudere) che sono CHIUSI in condizioni di riposo
  - num_contatti_istantanei_NO, num_contatti_ritardati_NO - numero dei contatti da comandare che sono APERTI in condizioni di riposo

    Gli aggettivi "istantanei" e "Ritardati" indicano rispettivamente l'apertura immediata o ritardata di un δt dei contatti

- **Attuatore, Sensore, Elemento_colloquio_uomo** Queste 3 classi rappresentano specializzazioni della classe 'Elemento primario'. Si differenziano solamente per alcuni attributi specifici.

  Gli attributi esprimono essenzialmente le caratteristiche elettriche del componente. Ad esempio, per la classe Attuatore si dichiarano i seguenti attributi:

[10] Vedi il paragrafo SIGLATURA in appedice per eventuali chiarimenti

  - num_cicli_lavoro_per_ora, num_ore_funz_al_giorno - Dati relativi al carico di lavoro a cui e' soggetto il motore
  - Categoria_impiego - Codice indicativo del tipo di impiego del motore secondo una normativa comunitaria

- **Filo** Questa classe raccoglie tutti i fili di collegamento
- **Contatto** Questa classe raccoglie tutti i contatti associati ai sottoelementi

Capitolo 5

Il Driver Object-World

5.1 Corrispondenze tra schema object-oriented e schema relazionale

Di seguito e' illustrata la maniera in cui ogni concetto della programmazione orientata agli oggetti venga tradotto nella visione relazionale.

- Ad ogni classe viene assegnata una tabella
- Ogni attributo semplice (cioe' definito su un dominio atomico) di una classe diventa un campo della tabella corrispondente
- Ogni singola istanza di una classe (oggetto) e' un record della tabella corrispondente
- L'identificatore di un oggetto (OID) in Visual C++ e' rappresentato dal puntatore all'oggetto stesso.
  Object-World mantiene una corrispondenza tra gli oggetti in memoria e i record nelle tabelle mediante l'uso di un codice univoco. Questo codice rappresenta sempre la chiave primaria in ogni tabella.
- Se un oggetto contiene un attributo, il cui dominio e' rappresentato da un'altra classe, tale attributo viene mappato in una chiave esterna. Object-World si occupa di trasferire la chiave esterna dell'oggetto referenziato senza costi aggiuntivi per il programmatore.
- Se un oggetto contiene un attributo multivalore (ad esempio set, list ecc...), esso viene implementato in Visual C++ mediante un array di puntatori. Tale attributo viene mappato in una tabella intermedia che permette di implementare una relazione molti-a-molti.

I metodi non trovano alcun concetto ad essi corrispondente nella visione relazionale, essi vengono conservati nel codice C++. Quando un oggetto viene reperito nel database, in memoria volatile ad esso vengono associati gli attributi e le proprie operazioni.

5.2 L'organizzazione del Driver

Object-World presenta una architettura molto semplice. Esso e' composto da tre moduli:

- una interfaccia verso il programmatore C++, rappresentata da un insieme di classi predefinite.
- un database relazionale (denominato **database di configurazione**) che associa le informazioni delle classi alle informazioni delle tabelle.
  Questo database conserva tutti i collegamenti tra la visione ad oggetti e quella relazionale. E' compito del programmatore che utilizza Object-World inserire le informazioni nel database.
- il database che funge da contenitore delle informazioni delle classi (denominato **database utente**) contenente tutte le tabelle che implementano il dominio applicativo.

In figura 5.1 viene presentata l'architettura di Object-World.

Object-World e' stato scritto in Visual C++ ed utilizza Microsoft ODBC.
I programmatori che usano ODBC in Visual C++ possono accedere ai dati esclusivamente mediante delle classi **CRecordSet**, le quali implementano delle istruzioni select di SQL. Nel medesimo modo il Driver Object-World permette di accedere ad insiemi di oggetti mediante delle classi denominate **CSelectOggetti**. Una classe CSelectOggetti incapsula al suo interno una raccolta di oggetti di una classe specificata dal programmatore.

5.3 Il database di configurazione

In figura 5.2 e' mostrato lo schema Object-Model della metodologia OMT, relativo al database di configurazione. I compiti del database di configurazione sono i seguenti:

- conservare l'elenco degli attributi delle classi della visione ad oggetti.

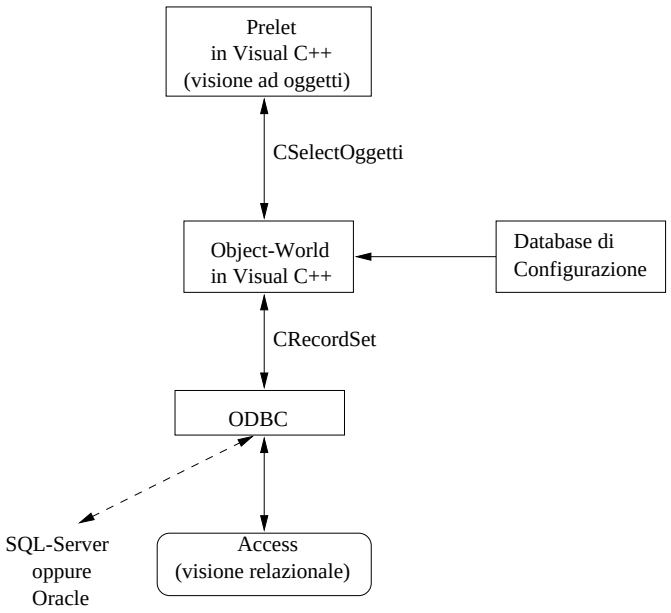

Figura 5.1: Architettura di Object-World

- conservare la categoria di ogni attributo, in modo che Object-world assuma un comportamento diverso in funzione del tipo di attributo con il quale opera.
- mettere in corrispondenza i nomi delle classi con i nomi delle tabelle della visione relazionale.
- conservare per ogni attributo multiplo :
  - la tabella intermedia che serve da appoggio al Driver
  - i nomi dei campi (campo1 e campo2) della tabella intermedia

Categorie degli attributi I campi di una tabella in una base di dati relazionale, oltre a differire per il tipo di valore che contengono, si differenziano anche per la loro funzione.
Possono rappresentare un semplice attributo oppure una chiave esterna, la quale mette il record in relazione con altre ennuple di altre tabelle.
Nel modello relazionale le possibili associazioni tra entita' sono le seguenti:

- uno-a-uno
- uno-a-molti o molti-a-uno

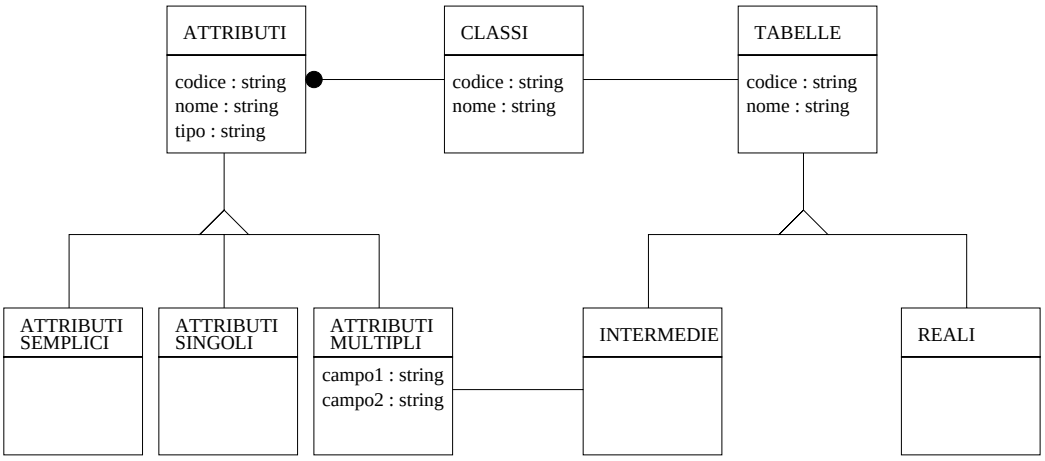

Figura 5.2: Objcet Model del database di configurazione

- molti-a-molti

In questo Driver gli attributi sono stati suddivisi in tre categorie, con le quali si riescono ad implementare le tre relazioni esistenti nel modello relazionale. Le categorie sono le seguenti:

- "attributi semplici" (non rappresentano alcuna relazione)
- "attributi singoli" (stabiliscono una relazione uno-a-uno, uno-a-molti o molti-a-uno)
- "attributi multipli" (stabiliscono una relazione molti-a-molti)

Questa suddivisione e' estremamente importante, in quanto il driver assume comportamenti diversi a seconda della categoria dell'attributo.

- Il valore contenuto negli attributi semplici viene semplicemente trasferito nelle rispettive tabelle.

- Il valore contenuto negli attributi singoli differisce dal corrispondente valore nel campo della tabella. Cio' poiche' un attributo singolo contiene il puntatore all'oggetto referenziato, mentre il corrispondente campo nella visione relazionale contiene la chiave esterna. Object-World e' in grado di eseguire autonomamente questa trasformazione di valore.

- Gli attributi multipli permettono di implementare insiemi di oggetti, quindi il loro valore viene trasformato in chiavi esterne da conservare in una tabella intermedia. Come e' noto infatti, nel mondo relazionale, e' possibile realizzare una relazioni molti-a-molti solamente mediante l'uso di una tabella intermedia.
  Object-World e' in grado di gestire questi legami senza che il programmatore si accorga dell'uso di una tabella di appoggio.

Tipi di dato Gli attributi degli oggetti da rendere persistenti non possono essere di tipo arbitrario. Il Driver prevede una categoria piuttosto ristretta di tipi utilizzabili. Sono previsti quattro tipi: "stringhe", "interi", "double" e "byte". Ognuno di questi e' rappresentato da una classe che ne contiene il valore e metodi per accedervi. Queste classi sono assolutamente trasparenti al programmatore che utilizza il Driver.

5.4 Descrizione tecnica di Object-World

Per poter rendere persistente una classe dichiarata in Visual C++ e' necessario:

- Compilare accuratamente il database di configurazione in modo che il driver sappia convertire correttamente tutte le informazioni associate alla classe
- Derivare la classe in oggetto dalla classe **CObjectPersistente**, predefinita all'interno del Driver

Per poter lavorare con classi arricchite con le potenzialita' di persistenza, e' possibile sfruttare i metodi offerti da Object-World, come ad esempio:

- Accedere agli attributi della classe mediante i metodi **LeggiValore** e **ScriviValore**
- Caricare e Salvare le istanze della classe attraverso i metodi **CaricaOggetto** e **SalvaOggetto**

L'uso dei suddetti metodi e' permesso solo all'interno di selezioni gia' aperte. Questo significa che non e' possibile creare degli oggetti con l'operatore new come avviene usualmente nel C++. Bensi' bisogna creare una CSelectOggetti sulla classe specificata, creare un nuovo oggetto con il metodo InserisciOggetto e salvare l'operato con SalvaOggetto.

Classi che rappresentano i dati Le classi che rappresentano i dati sono le seguenti: CTipo (classe base), CStringa, CIntero, CDoppio, CByte, CPuntatore, CMultiplo.
Mentre le prime quattro contengono solo un dato membro in cui salvare il valore del campo, le ultime due sono piu' complesse.

- La CPuntatore contiene, infatti, oltre ad un puntatore alla classe CTipo per il valore del campo, una stringa in cui salvare il nome della tabella referenziata e un puntatore a CObjectPersistente per poter stabilire il collegamento con l'oggetto con cui e' in relazione.

```
class CPuntatore : public CTipo
{
DECLARE_SERIAL(CPuntatore)
public:
CTipo *tipo;  //tipo di dato che rappresenta la chiave esterna
CString chiave_su;  //nome tabella da accedere con la query
CObjectPersistente *ele;  //puntatore all'oggetto referenziato

CPuntatore();
~CPuntatore();
virtual void DoFieldExchange(CString str,CDaoFieldExchange* pFX);
virtual void DoDataExchange(CString str,int nID,CDataExchange* pDX);
CPuntatore& operator=(const CPuntatore& elemento_sorgente);
CPuntatore(const CPuntatore& elemento_sorgente);
virtual void LeggiValore(CObjectPersistente* *campo);
virtual void ScriviValore(CObjectPersistente* campo);
virtual CObjectPersistente* LeggiPunt();
};
```

- La CMultiplo ha un array di oggetti per poter contenere i valori delle chiavi esterne e un array di puntatori agli oggetti referenziati. Possiede inoltre le stringhe che contengono il tipo di dato del campo, il nome della tabella referenziata, il nome della tabella intermedia che realizza la relazione e il nome dei campi della tabella intermedia.

```
class CMultiplo : public CTipo
{
DECLARE_SERIAL(CMultiplo)
public:
```

```
CObArray *elev;  //array di valori delle chiavi esterne
CPtrArray *eles;  //array di puntatori agli oggetti referenziati
CString tipo;  //descrizione del tipo di dato
CString chiave_su;  //tabella referenziata dall'attributo
CString intermedia;  //nome tabella intermedia
CString campo1;  //primo campo tabella intermedia
CString campo2;  //secondo campo tabella intermedia

CMultiplo();
~CMultiplo();
virtual void DoFieldExchange(CString str,CDaoFieldExchange* pFX);
virtual void DoDataExchange(CString str,int nID,CDataExchange* pDX);
CMultiplo& operator=(const CMultiplo& elemento_sorgente);
CMultiplo(const CMultiplo& elemento_sorgente);
virtual void LeggiValore(CObjectPersistente* *campo,int i);
virtual void SettaValore(CObjectPersistente* campo,int i);
virtual void CancellaValore(CObjectPersistente* campo);
virtual void CancellaValore(int i);
virtual void ModificaValore(CObjectPersistente* campo,int i);
virtual CObjectPersistente* LeggiMultiplo(int i);
};
```

Classi che rappresentano gli oggetti

- CObjectPersistente
  Ogni classe che si intende rendere persistente deve essere ereditata dalla classe CObjectPersistente. Tale classe eredita' tutte le caratteristiche della classe CObject delle Microsoft Foundation Class, inoltre aggiunge ad esse la capacita' di rendere persistente l'oggetto.

- CSelectOggetti
  Quando si vuole accedere al database contenente gli oggetti, bisogna creare un oggetto di CSelectOggetti e definire una interrogazione che abbia come risultato l'insieme di oggetti richiesti

Le funzioni del Driver

- **CSelectOggetti::DefinisciSelezione**

  Per poter accedere ai dati, la prima cosa da fare e' dichiarare un'istanza della classe CSelectOggetti e poi richiamare la funzione DefinisciSele-

zione. Questa funzione restituisce un numero intero che rappresenta il numero di oggetti (records) trovati. Il metodo DefinisciSelezione ha la seguente sintassi:

```
int DefinisciSelezione(CString nome_database_configurazione,
                       CString nome_file,
                       CString nclasse,
                       CString filtro = "",
                       CString ordine = "");
```

I suoi parametri sono cinque:

- I primi due riguardano i nomi dei file che contengono i due database: il database di configurazione e il database utente.
- Il terzo parametro e' il nome della classe sulla quale si intende eseguire la selezione.
- Il quarto e il quinto parametro sono, rispettivamente, le clausole WHERE e ORDER BY che si intendono applicare alla selezione in corso. Da notare come questio due parametri sono opzionali, se si vuole eseguire una selezione senza condizioni, la funzione accetta la stringa vuota come default.

Tramite questi parametri la funzione DefinisciSelezione puo' costruire una chiamata ad ODBC per connettersi al database relazionale, recuperare i dati e metterli negli attributi degli oggetti creati, mostrando in questo modo in una visione object-oriented.

- **CObjectPersistente::CaricaOggetto**

Questa funzione viene richiamata dalla funzione DefinisciSelezione, ma e' un metodo della classe CObjectPersistente. Essa deve riempire gli attributi degli oggetti con i valori corrispondenti del database relazionale.
Per quanto riguarda gli attributi semplici viene semplicemente recuperato il valore dal database. Per quanto riguarda gli attributi singoli, rappresentati da classi CPuntatore, viene caricato il valore della chiave esterna, ma non viene stabilito alcun collegamento tramite il puntatore. Per quanto riguarda gli attributi multipli, rappresentati da classi CMultiplo, vengono caricati tutti i valori contenuti nella tabella intermedia, in modo da conoscere con quali record si e' in relazione, ma, anche in questo caso, non vengono stabilite connessioni tramite i puntatori.

- **CSelectOggetti::InserisciOggetto**

  Questa funzione serve per preparare l'inserimento di un nuovo oggetto, il quale corrisponde ad un nuovo record nel database relazionale.

- **CObjectPersistente::SalvaOggetto**

  Questa funzione serve per aggiungere effettivamente nella base di dati l'oggetto creato poco prima con la InserisciOggetto.
  Gli attributi semplici vengono direttamente trasferiti nei campi della tabella corrispondente.
  Per gli attributi singoli viene trasferito il valore della chiave esterna.
  Per gli attributi multipli vengono trasferiti, nella tabella intermedia, le coppie di chiavi che realizzano la relazione molti-a-molti.

- **CObjectPersistente::AnnullaOggetto**

  Questa funzione serve per annullare le modifiche o gli inserimenti di oggetti nel database.

- **CObjectPersistente::ScriviValore**

  Questa funzione serve per poter impostare un valore in un determinato attributo dell'oggetto.

  - Se si tratta di un attributo semplice e' sufficiente passare ad essa il valore che si intende memorizzare. In questo caso la sintassi e' la seguente:

    ```
    ScriviValore(<nome attributo>,<valore>);
    ```

  - Nel caso si tratti di un attributo singolo, il valore e' rappresentato da un puntatore, precisamente il puntatore all'oggetto referenziato. Tramite questo puntatore Object-World puo' accedere al valore della chiave dell'oggetto, e tramite questa informazione esso e' in grado di memorizzare il legame uno-a-molti, sottoforma di chiave esterna, nel database relazionale.
    In questo caso la sintassi e' la seguente:

```
ScriviValore(<nome attributo>,<puntatore all'oggetto>);
```

– Nel caso che l'attributo sia multiplo, si segue lo stesso meccanismo degli attributi singoli, solamente bisogna tenere in considerazione che si lavora con un insieme di oggetti (al quale corrisponde un insieme di chiavi esterne). Quindi per la funzione e' necessario un ulteriore parametro che indichi quale oggetto dell'insieme si deve modificare. La sintassi e' la seguente:

```
ScriviValore(<nome attributo>,
                    <indice del puntatore>,
                    <puntatore all'oggetto>);
```

- **CObjectPersistente::LeggiValore**

  Per ogni funzione ScriviValore esiste la corrispondente funzione LeggiValore, la quale e' in grado di restituire:

  – il valore di un attributo semplice
  – il puntatore all'oggetto referenziato da un attributo singolo
  – uno dei puntatori contenuti in un attributo multiplo

- **CSelectOggetti::Chiudi**

  Questa funzione serve per chiudere la connessione al database e per rimuovere dalla memoria le strutture create per gestire la visione ad oggetti della base di dati.

- **Altre funzioni**

  Vi sono anche altre funzioni all'interno del driver, le quali servono per la navigazione attraverso gli oggetti della struttura.
  Esse sono: MuoviPrimoOggetto, MuoviUltimoOggetto, MuoviProssimoOggetto, MuoviPrecedenteOggetto, RestituisciOggettoCorrente, NumeroOggetti.

5.5 Limiti di Object-World

Rimangono molti aspetti che Object-World non considera, alcuni di questi sono elencati di seguito:

- la capacita' degli oggetti di riconoscere la gerarchia di cui fanno parte. Infatti, questa gerarchia, nel database di configurazione, e' stata temporaneamente appiattita per poter permettere al driver di cominciare a funzionare fin da subito. Questa soluzione ha il difetto di

presentare delle classi che non hanno una corrispondente tabella nel database relazionale. Questo sara' il prossimo grado di evoluzione di Object-World.

- Il driver non possiede un proprio linguaggio di interrogazione, quindi per eseguire delle query e' necessario utilizzare SQL, il quale costringe il programmatore a conoscere la realta' relazionale sottostante.
- Il driver non carica in memoria il contenuto degli oggetti referenziati, quindi per poter navigare nella struttura ad oggetti mediante dei join impliciti e' necessario eseguire ulteriori transazioni.

Per ulteriori informazioni riguardanti le problematiche di interfaccia tra visione object-oriented e DBMS relazionali si veda [BGSV97].

Capitolo 6

L'ambiente ODB-Tools preesistente

6.1 Architettura di ODB-Tools

Il progetto ODB-Tools ha come obiettivo lo sviluppo di strumenti per la progettazione assistita di basi di dati ad oggetti e l'ottimizzazione semantica di interrogazioni. Si basa su algoritmi che derivano da tecniche dell'intelligenza artificiale. Il risultato della ricerca svolta nell'ambito di questo progetto e' un prototipo che realizza l'ottimizzazione di schemi e l'ottimizzazione semantica delle interrogazioni.

In questa sezione intendo presentare in modo schematico ODB-Tools e i suoi principali componenti. Questo al fine di chiarire la sua struttura e permettere al lettore una maggiore comprensione dei miglioramenti apportati con il lavoro di questa tesi.

In figura 6.1 sono rappresentati i vari moduli che compongono tale prototipo. ODB-Tools e' composto da 3 moduli:

- ODL_TRASL (il traduttore):
  Dato uno schema di base di dati ad oggetti descritta in ODL-ODMG, scopo del traduttore e' generare la descrizione del medesimo schema in OCDL ed in formato `vf` (*Visual Form*) in modo che sia visualizzabile utilizzando l'applet Java `scvisual`.

- OCDL-designer:
  Questo componente software consente di controllare la consistenza di

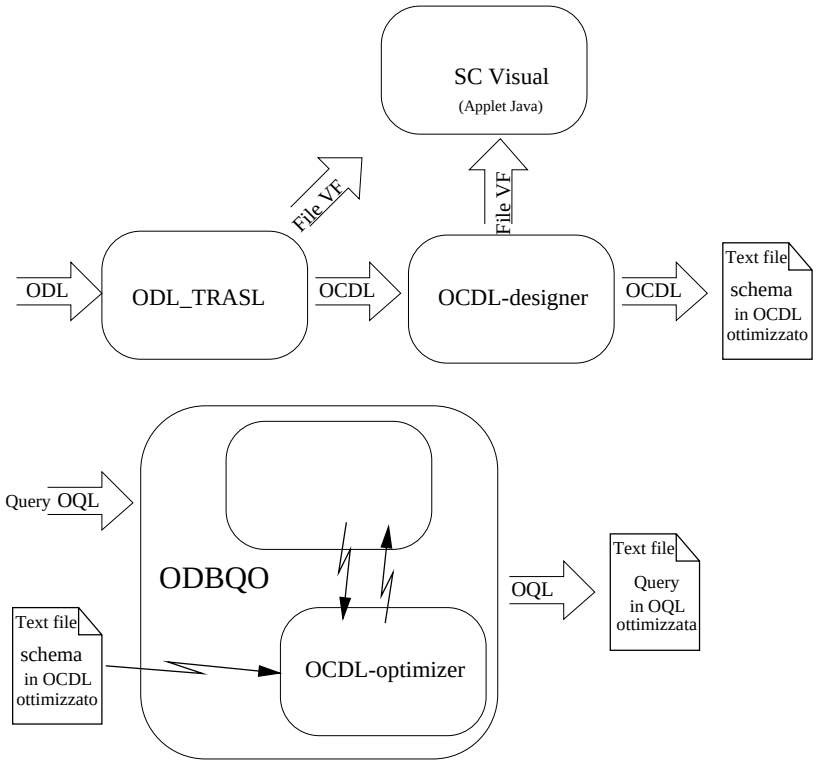

Figura 6.1: Componenti ODB-Tools

uno schema di base di dati ad oggetti e di ottenere la minimalita' dello schema rispetto alla relazione **isa**.

- ODBQOptimizer (ODBQO):
  Modulo adibito all'ottimizzazione semantica delle interrogazioni.

6.2 Il formalismo OCDL

In questa sezione viene brevemente riportato il formalismo OCDL introdotto in [BN94] ed esteso in [JBBV95] con i vincoli di integrità e vengono formalmente definite la sussunzione e l'espansione semantica.

6.2.1 Schema e Istanza del Database

Sia **D** l'insieme infinito numerabile dei valori atomici (che saranno indicati con $d_1, d_2, \ldots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani.

Sia $\mathbf{B}$ l'insieme di designatori di tipi atomici, con $\mathbf{B} = \{$`integer`, `string`, `bool`, `real`, i_1–j_1, i_2–j_2, ... , $d_1, d_2, \ldots\}$, dove i d_k indicano tutti gli elementi di `integer`$\cup$`string`$\cup$`bool` e dove gli i_k–j_k indicano tutti i possibili intervalli di interi (i_k può essere $-\infty$ per denotare il minimo elemento di `integer` e j_k può essere $+\infty$ per denotare il massimo elemento di `integer`).

Sia $\mathbf{A}$ un insieme numerabile di *attributi* (denotati da $a_1, a_2, \ldots$) e $\mathcal{O}$ un insieme numerabile di *identificatori di oggetti* (denotati da $o, o', \ldots$) disgiunti da $\mathbf{D}$. Si definisce l'insieme $\mathcal{V}(\mathcal{O})$ dei *valori su* $\mathcal{O}$ (denotati da v, v') come segue (assumendo $p \geq 0$ e $a_i \neq a_j$ per $i \neq j$):

$$v \ \rightarrow d \mid o \mid \{v_1, \ldots, v_p\} \mid [a_1 : v_1, \ldots, a_p : v_p]$$

Gli identificatori di oggetti sono associati a valori tramite una *funzione totale* δ da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$; in genere si dice che il valore $\delta(o)$ è lo *stato* dell'oggetto identificato dall'l'oid o;

Sia $\mathbf{N}$ l'insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che $\mathbf{A}$, $\mathbf{B}$, e $\mathbf{N}$ siano a due a due disgiunti. $\mathbf{N}$ è partizionato in tre insiemi $\mathbf{C}$, $\mathbf{V}$ e $\mathbf{T}$, dove $\mathbf{C}$ consiste di nomi per *tipi-classe base* ($C, C' \ldots$), $\mathbf{V}$ consiste di nomi per *tipi-classe virtuali* ($V, V' \ldots$), e $\mathbf{T}$ consiste di nomi per *tipi-valori* ($t, t', \ldots$).

Un *path* p è una sequenza di elementi $p = e_1.e_2.\ldots.e_n$, con $e_i \in \mathbf{A} \cup \{\triangle, \forall, \exists\}$. Con ϵ si indica il path vuoto.

$\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$[1] indica l'insieme di tutte le *descrizioni di tipo finite* (S, S', ...), dette brevemente *tipi*, su di un dato $\mathbf{A}, \mathbf{B}, \mathbf{N}$, ottenuto in accordo con la seguente regola sintattica:

$$S \ \rightarrow \ \top \mid B \mid N \mid [a_1 : S_1, \ldots, a_k : S_k] \mid \forall\{S\} \mid \exists\{S\} \mid \ \triangle S \mid S \sqcap S' \mid (p : S)$$

$\top$ denota il *tipo universale* e rappresenta tutti i valori; [ ] denota il costruttore di tupla. $\forall\{S\}$ corrisponde al comune costruttore di insieme e rappresenta un insieme i cui elementi sono *tutti* dello stesso tipo S. Invece, il costruttore $\exists\{S\}$ denota un insieme in cui *almeno* un elemento è di tipo S. Il costrutto $\sqcap$ indica la *congiunzione*, mentre $\triangle$ è il costruttore di oggetto. Il tipo (p: S) è detto *tipo path* e rappresenta una notazione abbreviata per i tipi ottenuti

[1] In seguito, scriveremo $\mathbf{S}$ in luogo di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ quando i componenti sono ovvi dal contesto.

con gli altri costruttori.

Dato un dato sistema di tipi $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, uno *schema* σ su $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ è una funzione totale da $\mathbf{N}$ a $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, che associa ai nomi di tipi la loro descrizione. Diremo che un nome di tipo N *eredita* da un altro nome di tipo N', denotato con $N \prec\!\!\!\mid_\sigma N'$, se $\sigma(N) = N' \sqcap S$. Si richiede che la relazione di ereditarietà sia priva di cicli, i.e., la chiusura transitiva di $\prec\!\!\!\mid_\sigma$, denotata $\prec$, sia un ordine parziale stretto.

Dato un dato sistema di tipi $\mathbf{S}$, una *regole di integrità* R è espressa nella forma $R = S^a \rightarrow S^c$, dove S^a e S^c rappresentano rispettivamente l'antecedente e il conseguente della regola R, con $S^a, S^c \in \mathbf{S}$. Una regola R esprime il seguente vincolo: per tutti gli oggetti v, se v è di tipo S^a allora v deve essere di tipo S^c. Con $\mathbf{R}$ si denota un insieme finito di regole.

Uno *schema con regole* è una coppia $(\sigma, \mathbf{R})$, dove σ è uno schema e $\mathbf{R}$ un insieme di regole.

La *funzione interpretazione* $\mathcal{I}$ è una funzione da $\mathbf{S}$ a $2^{\mathcal{V}(\mathcal{O})}$ tale che: $\mathcal{I}[\top] = \mathcal{V}(\mathcal{O})$, $\mathcal{I}[B] = \mathcal{I}_{\mathbf{B}}[B]$[2] , $\mathcal{I}[C] \subseteq \mathcal{O}, \mathcal{I}[V] \subseteq \mathcal{O}, \mathcal{I}[t] \subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O}$. L'interpretazione è estesa agli altri tipi come segue:

$$
\begin{aligned}
\mathcal{I}[[a_1\colon S_1, \ldots, a_p\colon S_p]] &= \Big\{ [a_1\colon v_1, \ldots, a_q\colon v_q] \Big| p \leq q, v_i \in \mathcal{I}[S_i], 1 \leq i \leq p, \\
&\qquad\qquad v_j \in \mathcal{V}(\mathcal{O}), p{+}1 \leq j \leq q \Big\} \\
\mathcal{I}[\forall\{S\}] &= \Big\{ \{v_1, \ldots, v_p\} \Big| v_i \in \mathcal{I}[S], 1 \leq i \leq p \Big\} \\
\mathcal{I}[\exists\{S\}] &= \Big\{ \{v_1, \ldots, v_p\} \Big| \exists i, 1 \leq i \leq p, v_i \in \mathcal{I}[S] \Big\} \\
\mathcal{I}[\triangle S] &= \Big\{ o \in \mathcal{O} \Big| \delta(o) \in \mathcal{I}[S] \Big\} \\
\mathcal{I}[S \sqcap S'] &= \mathcal{I}[S] \cap \mathcal{I}[S']
\end{aligned}
$$

Per i tipi cammino abbiamo $\mathcal{I}[(p\colon S)] = \mathcal{I}[(e\colon (p'\colon S))]$ se $p = e.p'$ dove

$$\mathcal{I}[(\epsilon\colon S)] = \mathcal{I}[S],\ \mathcal{I}[(a\colon S)] = \mathcal{I}[[a\colon S]],\ \mathcal{I}[(\triangle\colon S)] = \mathcal{I}[\triangle S],$$

$$\mathcal{I}[(\forall\colon S)] = \mathcal{I}[\forall\{S\}],\ \mathcal{I}[(\exists\colon S)] = \mathcal{I}[\exists\{S\}]$$

[2]Assumendo $\mathcal{I}_{\mathbf{B}}$ funzione di *interpretazione standard* da $\mathbf{B}$ a $2^{\mathbf{B}}$ tale che per ogni $d \in \mathbf{D} : \mathcal{I}_{\mathbf{B}}[d] = \{d\}$.

Si introduce ora la nozione di istanza legale di uno schema con regole come una interpretazione nella quale un valore istanziato in un nome di tipo ha una descrizione corrispondente a quella del nome di tipo stesso e dove sono valide le relazioni di inclusioni stabilite tramite le regole.

Definizione 1 (Istanza Legale) *Una funzione di interpretazione* $\mathcal{I}$ *è una* istanza legale *di uno schema con regole* $(\sigma, \mathbf{R})$ *sse l'insieme* $\mathcal{O}$ *è* finito *e per ogni* $C \in \mathbf{C}, V \in \mathbf{V}, t \in \mathbf{T}, R \in \mathbf{R}$ *:* $\mathcal{I}[C] \subseteq \mathcal{I}[\sigma(C)]$, $\mathcal{I}[t] = \mathcal{I}[\sigma(t)]$, $\mathcal{I}[V] = \mathcal{I}[\sigma(V)]$, $\mathcal{I}[S^a] \subseteq \mathcal{I}[S^c]$.

Si noti come, in una istanza legale $\mathcal{I}$, l'interpretazione di un nome di classe base *è contenuta* nell'interpretazione della sua descrizione, mentre per un nome di classe virtuale, come per un nome di tipo-valore, l'interpretazione *coincide* con l'interpretazione della sua descrizione. In altri termini, mentre l'interpretazione di una classe base è fornita dall'utente, l'interpretazione di una classe virtuale è calcolata sulla base della sua descrizione.

In questa sede si suppone che lo schema con regole sia privo di cicli. Formalmente, diremo che un nome di tipo N *dipende* dal nome di tipo N' se N' appare in $\sigma(N)$ **oppure** se esiste in $\mathbf{R}$ una regola $N \to S$ e N' appare in S. Uno schema con regole e' privo di cicli se la relazione *dipende* non contiene cicli.

6.2.2 Sussunzione ed Espansione Semantica di un tipo

Introduciamo la relazione di sussunzione in uno schema con regole.

Definizione 2 (Sussunzione) *Dato uno schema con regole* $(\sigma, \mathbf{R})$, *la* relazione di sussunzione *rispetto a* $(\sigma, \mathbf{R})$, *scritta* $S \sqsubseteq_{\mathrm{R}} S'$ *per ogni coppia di tipi* $S, S' \in \mathbf{S}$, *è data da:* $S \sqsubseteq_{\mathrm{R}} S'$ *sse* $\mathcal{I}[S] \subseteq \mathcal{I}[S']$ *per tutte le istanze legali* $\mathcal{I}$ *di* $(\sigma, \mathbf{R})$.

Segue immediatamente che $\sqsubseteq_{\mathrm{R}}$ è un preordine (i.e., transitivo e riflessivo ma antisimmetrico) che induce una *relazione di equivalenza* $\simeq_{\mathrm{R}}$ sui tipi: $S \simeq_{\mathrm{R}} S'$ sse $S \sqsubseteq_{\mathrm{R}} S'$ e $S' \sqsubseteq_{\mathrm{R}} S$. Diciamo, inoltre, che un tipo S è *inconsistente* sse $S \simeq_{\mathrm{R}} \bot$, cioè per ciascun dominio l'interpretazione del tipo è sempre vuota.

È importante notare che la relazione di sussunzione rispetto al solo schema σ, cioè considerando $\mathbf{R} = \emptyset$, denotata con $\sqsubseteq_\sigma$, è simile alle relazioni di *subtyping* o *refinement* tra tipi definite nei *CODMs* [Atz93, LR89]. Questa relazione può essere calcolata attraverso una comparazione sintattica sui tipi; per il nostro modello tale l'algoritmo è stato presentato in [BN94].

Definizione 3 (Espansione Semantica) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *e un tipo* $S \in \mathbf{S}$, *l'*espansione semantica *di* S *rispetto a* $\mathbf{R}$, *EXP(S), è un tipo di* $\mathbf{S}$ *tale che:*

1. $EXP(S) \simeq_{\mathrm{R}} S$;
2. *per ogni* $S' \in \mathbf{S}$ *tale che* $S' \simeq_{\mathrm{R}} S$ *si ha* $EXP(S) \sqsubseteq_\sigma S'$.

In altri termini, EXP(S) è il tipo più specializzato (rispetto alla relazione $\sqsubseteq_\sigma$) tra tutti i tipi $\simeq_{\mathrm{R}}$–equivalenti al tipo S. L'espressione $EXP(S)$ permette di esprimere la relazione esistente tra $\sqsubseteq_{\mathrm{R}}$ e $\sqsubseteq_\sigma$: per ogni $S, S' \in \mathbf{S}$ si ha $S \sqsubseteq_{\mathrm{R}} S'$ se e solo se $EXP(S) \sqsubseteq_\sigma S'$. Questo significa che, dopo aver determinato l'espansione semantica, anche la relazione di sussunzione nello schema con regole può essere calcolata tramite l'algoritmo presentato in [BN94].
È facile verificare che, per ogni $S \in \mathbf{S}$ e per ogni $R \in \mathbf{R}$, se $S \sqsubseteq_\sigma (p : S^a)$ allora $S \sqcap (p\colon S^c) \simeq_{\mathrm{R}} S$. Questa trasformazione di S in $S \sqcap (p\colon S^c)$ è la base del calcolo della $EXP(S)$: essa viene effettuata iterativamente, tenendo conto che l'applicazione di una regola può portare all'applicazione di altre regole. Per individuare tutte le possibili trasformazioni di un tipo implicate da uno schema con regole $(\sigma, \mathbf{R})$, si definisce la funzione totale $, : \mathbf{S} \longrightarrow \mathbf{S}$, come segue:

$$, (S) \;=\; \begin{cases} S \sqcap (p : S^c) & \text{se esistono } R \text{ e } p \text{ tali che } S \sqsubseteq_\sigma (p\colon S^a) \text{ e } S \not\sqsubseteq_\sigma (p\colon S^c) \\ S & \text{altrimenti} \end{cases}$$

e poniamo $,\tilde{} = , ^i$, dove i è il più piccolo intero tale che $, ^i = , ^{i+1}$. L'esistenza di i è garantita dal fatto che il numero di regole è finito e una regola non può essere applicata più di una volta con lo stesso cammino ($S \not\sqsubseteq_\sigma (p\colon S^c)$). Si può dimostrare che, per ogni $S \in \mathbf{S}$, EXP(S) è effettivamente calcolabile tramite $,\tilde{}\,(S)$.

6.3 Pregi e limiti espressivi di ODB-Tools

La struttura del database e' stata espressa in ODL esteso (vedi appendice A.1).
Durante la fase di progettazione dello schema del database del prototipo "Elet-Designer" si e' potuto verificare il funzionamento di ODB-Tools ed individuare, oltre ai suoi aspetti positivi, i suoi limiti.
In questo paragrafo si intendono evidenziare i vari aspetti di ODB-Tools specificando, per i problemi incontrati, le soluzioni adottate.

Pregi

- La parte piu' interessante riguarda i vincoli di integrita' esprimibili come regole "if ... then ...".

In realta' il database in oggetto ha necessita' di esprimere molti vincoli, questo perche' riguarda elementi elettrici connessi tra loro che hanno per loro natura un legame molto stretto. Questo ha permesso di esprimere vincoli di diversa natura, ad esempio:

- vincoli di dominio
- vincoli di integrita' referenziale
- vincoli che legano attributi della stessa classe, come ad esempio "if then" che si usano in ODB-Tools
- vincoli che legano attributi complessi (come set, list ecc..)
- vincoli che legano attributi di classi diverse: in questo caso si deve operare utilizzando la navigazione mediante la "dot notation"

Tutti questi vincoli sono riconducibili a una forma interpretabile da ODB-Tools.

- ODB-Tools mette a disposizione un insieme piu' ampio di tipi di quello disponibile in Visual C++, Questo permette di esprimere piu' facilmente i tipi degli attributi in fase di progettazione, ma questo vantaggio si perde totalmente nella fase di implementazione in Visual C++. Ad esempio, nel prototipo "Elet-Designer" si sono dovuti tradurre tutti i tipi range in tipi integer, in quanto il Visual C++ non dispone di un tipo range predefinito.

- ODB-Tools evidenzia gli errori che si verificano nel susseguirsi di modifiche allo schema di un database, durante le prime fasi di progettazione. Di seguito si riporta uno schema errato e l'output presentato da ODB-Tools:

```
interface Attuatore : Elemento_primario
( extent Attuatori
)
{
  attribute real potenza;
  attribute range {0.0,1.0} cosfi;
  attribute integer num_cicli_lavoro_per_ora;
  attribute range {0,24} num_ore_funz_al_giorno;
  attribute range{1,8} categoria_impiego;
  attribute real coefficiente_contemporaneita;
};
rule Att2 forall X in Attuatore:
```

```
                X.alimentato.tipo_alimentazione="AC"
  then X.categoria_impiego>5 and X.categoria_impiego<9;
```

OCDL-Designer presenta il seguente output, nel quale viene evidenziata una incoerenza:

```
*************** OCDL-DESIGNER ***************
Source Schema Acquisition: OK
WARNING inchoerent class 'Att2c' detect
Canonical Form: OK
Subsumption: OK
Writing Output Files (impianto.fc,impianto.vf): OK
*********************************************
```

- ODB-Tools evidenzia regole superflue, permettendo al progettista di rendere piu' semplice lo schema del database.

Limiti espressivi di OCDL

- Dall'analisi del dominio applicativo sono risultati dei vincoli di integrita' piu' complessi.
  Innanzitutto ODB-Tools permette di inserire in una regola condizioni del tipo :

  ⟨attributo⟩⟨operatore⟩⟨valore⟩

  ossia e' possibile mettere in relazione un campo con un valore costante (definito nella regola). Questo e' corretto, perche' non avrebbe senso a livello intensionale confrontare due attributi, e non avrebbe nemmeno senso presentare questa situazione all'algoritmo di sussunzione.

  In realta' il problema sollevato in questa tesi riguarda la presenza di vincoli (o regole) da controllare a livello estensionale: ad esempio affermare che "la corrente di un circuito deve essere uguale alla somma delle correnti degli elementi primari (ad esempio motori) alimentati dal circuito stesso" si traduce in un confronto tra un campo e una somma di campi da effettuarsi esclusivamente a livello estensionale (ovvero quando il database viene utilizzato nel software).

- Si riscontra l'impossibilita' di esprimere legami complessi tra gli attributi, come ad esempio la moltiplicazione, la divisione tra campi del database.

- Si riscontra l'impossibilita' di esprimere il comportamento nella descrizione di una classe, attraverso la dichiarazione di metodi. Questo problema e' stato risolto estendendo ulteriormente la sintassi di ODL e permettendo l'inserimento di metodi, sia nella interfaccia delle classi che nei predicati delle rule.

- Si riscontra la mancanza dell'operatore logico "OR" all'interno dei predicati delle rule. Questo limite e' stato superato introducendo una rule per ogni clausola del predicato OR.

- Il set di tipi base, pur essendo piu' ampio del Visual C++, risulta essere un po' ristretto: infatti non e' possibile esprimere range di valori numerici non interi. Questo e' un forte limite perche' molti attributi dello schema del database sono dei tipi 'float' che rappresentano delle correnti elettriche. Tutti questi attributi devono essere dichiarati 'float' senza avere la possibilita' di vincolarli con un range di valori reali.
  Questa limitazione e' stata superata introducendo il tipo range di float. Il tipo "range di float" presenta la medesima sintassi del tipo "range di interi", e' solamente necessario introdurre valori contenenti il punto decimale in modo da informare il sistema sul corretto tipo che si intende utilizzare.

- Non e' disponibile il tipo enumerato. Per superare questa limitazione gli attributi di tipo enumerato sono stati trasformati in range di interi, in cui ogni valore intero rappresenta un elemento del tipo enumerato corrispondente.
  Esempi di attributi dichiarati in questo modo sono i seguenti:

  - Attuatore::categoria_impiego
  - Circuito::tensione
  - Contatto::tipo_contatto

- Un ulteriore limite e' rappresentato dalla mancanza di un costruttore che permetta di esprimere il concetto di generalizzazione. Mediante ODL-esteso e' possibile indicare che una classe eredita le proprieta' da altre classi gia' esistenti nello schema. Non vi sono strumenti per dichiarare che una classe e' la generalizzazione di altre classi gia' esistenti

nello schema. Strumenti del genere possono essere molto utili nelle fasi di evoluzione di uno schema.

Alcune delle limitazioni piu' significative sono state affrontate e risolte dalla estensioni di OCDL e di ODB-Tools introdotte nel capitolo 7.

Capitolo 7

Progetto di estensioni di ODB-Tools

In questo capitolo viene illustrata l'estensione del modello OCDL per:

- supportare la descrizione di interfacce di metodi
- supportare la descrizione di regole con predicati che contengono metodi

Come esempio, i nuovi concetti introdotti verranno applicati alla modellazione di un impianto elettrico. Viene inoltre presentata la conseguente estensione di ODB-Tools ed in particolare di due componenti :

- il modulo traduttore (odl_trasl)
- il componente di OCDL-Designer che si occupa di redigere la forma canonica dello schema.

Allo scopo di estendere il modello OCDL con le operazioni, si analizzano alcune proposte di rappresentazione della semantica delle operazioni e di controllo sulle interfaccie dei metodi.
L'obiettivo è quello di individuare quali elementi possono essere integrati nel modello OCDL e nel sistema ODB-Tools.
In generale la presenza delle operazioni nella parte dichiarativa non e' mai ricca di significato. Questo e' facile da capire, infatti non sono molte le informazioni che si possono dichiarare nella signature di una operazione.
La maggior parte dei sistemi attuali dichiarano ad esempio :

- il tipo di ritorno
- il nome della operazione

- il numero di parametri
- il nome di ogni parametro
- il tipo di ogni parametro

Alcuni tentativi per rappresentare, a livello dichiarativo, la semantica delle operazioni, sono state presentate in letteratura:

- In ODMG-93 [Cat94] e [ADF+94] e' stata introdotta per ogni parametro la possibilita' di indicarne l'attributo (**in**, **out** o **inout**) attraverso l'utilizzo di una parola chiave da anteporre al parametro stesso. Questa informazione ci permette di capire l'utilizzo del parametro all'interno della operazione e quindi e' estremamente utile renderla disponibile gia' in sede dichiarativa. Questo aspetto verrà analizzato nel paragrafo 7.1. Nello stesso paragrafo il modello OCDL verrà esteso con l'introduzione delle operazioni.

- Per quello che riguarda il controllo di consistenza effettuato sulla interfaccia dei metodi, nel paragrafo 7.3 si analizzeranno i due approcci presentati in letteratura, noti con il nome di *principio di covarianza* e *principio di controvarianza*. Nello stesso paragrafo il modello OCDL verrà esteso con la definizione formale di covarianza e controvarianza.

- Per quello che riguarda la semantica delle operazioni, nel paragrafo 7.4 viene presentato un lavoro teorico, fatto con riferimento al modello O_2 [BDK96], per la rappresentazione del flusso di controllo del codice di una operazione e il conseguente controllo di consistenza. Siccome tale controllo è essenzialmente basato sulla gerarchia di ereditarietà delle classi, l'obiettivo è stato quello di uno studio preliminare sull'applicabilità in tale ambito formale del concetto di sussunzione.

7.1 Introduzione delle operazioni

Operazioni all'interno delle *interface* delle classi La sintassi che esprime la signature delle operazioni all'interno della interface delle classi e' la medesima di ODMG 93 [ADF+94].

< OpDcl > ::= < OpTypeSpec > < Identifier > < ParameterDcls >
< ParameterDcls > ::= (< ParamDclList >) | ( )
< ParamDclList > ::= < ParamDcl > |
< ParamDcl > , < ParamDclList >
< ParamDcl > ::= [< ParamAttribute >] < SimpleTypeSpec > < Declarator >
< ParamAttribute > ::= **IN** | **OUT** | **INOUT**

La spiegazione della suddetta sintassi e' contenuta nei documenti ufficiali di CORBA [For93], infatti il linguaggio ODL-ODMG e' stato definito partendo dalle primitive proposte in IDL (Interface Definition Language), il linguaggio descrittivo di CORBA.

Con il termine "attributo del parametro" (ParamAttribute) si intende esprimere la direzione del parametro. Infatti CORBA presuppone di lavorare in ambiente client-server e indica tre possibili direzioni del parametro:

1. **in** - il parametro viene passato dal client al server, quindi esso e' un parametro di solo input per l'operazione. Questo attributo viene assegnato di default al parametro nel caso non venga specificato nessun attributo.

2. **out** - il parametro viene passato dal server al client, quindi esso e' un parametro di solo output per l'operazione.

3. **inout** - il parametro viene passato in entrambe le direzioni, quindi esso e' un parametro di input-output per l'operazione.

In questa tesi si utilizzano questi concetti, introdotti per gli ambienti client-server, per arricchire la signature delle operazioni e di conseguenza offrire al sistema ODB-Tools maggiori informazioni relative alla operazione stessa.
Un esempio di operazione dichiarata all'interno della interface della classe e' il seguente:

```
// --------- Circuito
interface Circuito : Elemento_primario
( extent Circuiti
  keys codice_circuito
)
```

```
{
  attribute string tipo_linea;
  attribute string tipo_circuito;
  attribute string tipo_alimentazione;
  attribute range{1,6} tensione;
  attribute range{1,2} frequenza;
  relationship set<Elemento_elettrico> alimenta
       inverse Elemento_elettrico::alimentato;

  range{0,100} calcola_corrente_circuito
               (in set<Elemento_elettrico> alimenta);

};
```

Nel paragrafo 7.3.1 verra' introdotto il controllo di consistenza sulla signature delle operazioni.

Operazioni all'interno delle *rule* Con riferimento all'esempio "la corrente di un circuito deve essere uguale alla somma delle correnti degli elementi primari alimentati dal circuito stesso", riportato nel paragrafo 6.3, si e' pensato di esprimere "la somma delle correnti degli elementi primari alimentati dal circuito" come un **metodo** della classe (Circuito) a cui appartiene (da questo momento in poi parleremo di **operazioni** seguendo l'indicazione dello standard ODMG 93).

Le operazioni dichiarate all'interno delle rule possono appartenere solamente ad un sottoinsieme delle operazioni. Esse sono considerate delle funzioni matematiche. Questo significa che sono vincolate a rispettare le seguenti regole:

- devono restituire un risultato appartenente ad un tipo base
- devono possedere almeno un parametro
- ogni parametro deve avere attributo IN. Infatti non e' permesso indicare l'attributo del parametro nel caso si tratti di funzioni

Un esempio di operazione dichiarata all'interno di una rule e' il seguente:

```
// --------- Circuito
interface Circuito : Elemento_primario
( extent Circuiti
```

```
  keys codice_circuito
)
{
  attribute string tipo_linea;
  attribute string tipo_circuito;
  attribute string tipo_alimentazione;
  attribute range{1,6} tensione;
  attribute range{1,2} frequenza;
  relationship set<Elemento_elettrico> alimenta
       inverse Elemento_elettrico::alimentato;
};

rule Cir3 forall X in Circuito: X in Circuito
     then  X.corrente_assorbita_per_mille=
           range {0,100} calcola_corrente_circuito(X.alimenta);
```

In questo caso si e' dichiarata la medesima operazione per la classe Circuito, solamente essa e' presente all'interno della rule "Cir3". Per questo motivo non e' necessario indicare l'attributo e il tipo del parametro.

Poiche' OCDL-Designer ragiona sulla compatibilita' di tipo, in un predicato contenente una operazione e' possibile effettuare un controllo di consistenza solamente sul tipo del valore ritornato dal metodo.

7.2 Estensione di OCDL con operazioni

7.2.1 Introduzione di uno schema di operazioni

Nel paragrafo 6.2 e' stata presentata la logica OCDL su cui e' gia' basato ODB-Tools, che esprime solo descrizioni strutturali. Viene ora presentata l'integrazione delle operazioni.

Sia **N** l'insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che **A**, **B**, e **N** siano a due a due disgiunti. **N** è partizionato in tre insiemi **C**, **V** e **T**, dove **C** consiste di nomi per *tipi-classe base* ($C, C' \ldots$), **V** consiste di nomi per *tipi-classe virtuali* ($V, V' \ldots$), e **T** consiste di nomi per *tipi-valori* ($t, t', \ldots$).

Si introducono ora i tipi atti a rappresentare lo schema delle operazioni.

Sia **Att**={in,out,inout} l'insieme dei nomi dei tipi che indicano l' "attributo del parametro", (denotati da Att_1,Att_2,... e chiamati brevemente "tipi-parametro").
Tale insieme deve essere chiaramente disgiunto dall'insieme dei nomi dei tipi **N**.
I tipi-parametro servono per introdurre nel sistema dei tipi **S** il tipo "signature di operazione", denotato con S_s, che è un particolare tipo record:

$$[p_1{:}S_1 \sqcap \mathbf{Att}_1, \; p_2{:}S_2 \sqcap \mathbf{Att}_2, \ldots, p_k{:}S_k \sqcap \mathbf{Att}_k]$$

dove:
k è il numero dei parametri della operazione
p_1,p_2,...,p_k sono i nomi dei parametri formali della operazione
$S_i \in \mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ $\forall$**i=1,...k** sono i tipi dei parametri formali

Il tipo signature rappresenta interamente gli argomenti della signature di una operazione.

Nel seguito, il sistema dei tipi costruito partendo dall'insieme dei nomi dei tipi **N** unito con **Att** ed esteso con il tipo "signature di operazione" verrà, per semplicità, indicato ancora con $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$.

Per i tipi-parametro si deve estendere la definizione di interpretazione e di istanza; in particolare, per estendere il concetto di istanza, occorre estendere la funzione σ anche a tali nomi. Queste estensioni sono basate sul fatto che un tipo-parametro deve essere considerato un nome di tipo (sia classe che tipo-valore) primitivo e senza alcuna descrizione, quindi

- Interpretazione:

$$\mathcal{I}[\mathbf{Att}] \subseteq \mathcal{V}(\mathcal{O})$$

- Schema:

$$\sigma(\mathbf{Att}) = \top$$

- Istanza:

$$\mathcal{I}[\sigma(\mathbf{Att})] \subseteq \mathcal{I}[\mathbf{Att}]$$

Si introduce ora la definizione di operazione; informalmente, le operazioni sono rappresentate da quadruple contenenti:

1. la classe sulla quale sono definite
2. il nome
3. la signature, rappresentata da un tipo simile alle ennuple con l'aggiunta degli attributi dei parametri, anch'essi rappresentati da tipi
4. il tipo di ritorno

Definizione 4 (Operazione) *Una Operazione è una quadrupla* $(C_d,\mathbf{m},T_s,\mathbf{R})$, *dove:*
$C_d \in \mathbf{C} \cup \mathbf{V}$ *rappresenta la classe su cui e' definita l'operazione*
$\mathbf{m} \in \mathbf{M}$ *e' il nome della operazione*
$T_s \in \mathbf{S_s}$ *e' il tipo della signature della operazione*
$\mathbf{R} \in \mathbf{S}$ *e' il tipo del risultato della operazione*

I linguaggi di programmazione orientati agli oggetti prevedono la possibilita', nota come "overloading", di dichiarare due operazioni definite sulla medesima classe con lo stesso nome. Per permettere l'overloading la coppia nome e classe di definizione non sono sufficiente ad identificare una operazione. E' necessario aggiugere l'intera signature della operazione per poterla identificare.

Per generalizzare il discorso anche al caso in cui una operazione venga introdotta direttamente in una rule (problema affrontato nella prossima sezione) è stata fatta la seguente scelta: per ogni operazione dichiarata nella interface delle classi il sistema crea un identificatore univoco, detto *identificatore di operazione*, e vi associa la quadrupla che rappresenta l'operazione stessa. Si noti che una stessa operazione (stessa quadrupla) può essere associata a più identificatori.

Formalmente, si introduce $\mathbf{I_M}$, l'insieme numerabile degli identificatori delle operazioni (denotati da Op_1,Op_2,...) e si rappresenta questa corrispondenza tramite il concetto di *Schema di operazioni*.

Definizione 5 (Schema di operazioni) *Uno* schema di operazioni $\sigma_{\mathbf{M}}$ *su* $\mathbf{I_M}$ *è una funzione totale*
$\sigma_{\mathbf{M}} : \mathbf{I_M} \rightarrow (\mathbf{C} \cup \mathbf{V})\ X\ \mathbf{M}\ X\ \mathbf{S_s}\ X\ (\mathbf{S})$
che associa ad ogni identificatore la descrizione della propria operazione.

7.2.2 Introduzione di un identificatore di operazione

Nel paragrafo precedente e' stato introdotto lo schema di operazioni. Quando viene dichiarata una operazione all'interno di una rule, essa viene inserita nello schema delle operazioni, mediante una quadrupla ed un identificatore, come avviene per le operazioni dichiarate all'interno delle interface delle classi.

Esiste una differenza fondamentale tra le operazioni dichiarate all'interno della interface delle classi e quelle dichiarate all'interno delle rule. All'interno delle rule le operazione vengono "invocate", viene utilizzata una loro istanza. In generale esistono infinite possibilita' di invocare una operazione, eguali a tutti i possibili valori assunti dai parametri in ingresso alla operazione stessa. E' necessario distinguere il tipo di ritorno della operazione in base alla chiamata effettuata, poiche' nel momento in cui la rule sara' eseguita si otterranno risultati diversi. Per fare questo e' necessario introdurre un nome che appartenga al sistema dei tipi che univocamente rappresenti quella chiamata. Questo e' possibile realizzarlo utilizzando gli identificatori di operazioni definiti nel precedente paragrafo. Utilizzando gli identificatori di operazioni all'interno delle rule, e ponendosi l'obiettivo di mantenere distinte i diversi usi delle medesime operazioni si ottiene una funzione $\sigma_{\mathbf{M}}$ che mette due diversi identificatori in corrispondenza con la medesima quadrupla.

In una rule contenente un'operazione si deve affrontare anche il problema della sua traduzione in un tipo del modello OCDL. Nel seguito viene presentato un esempio per chiarire quali informazioni dell'operazione vengono riportate nella rule tradotta e quali informazioni vengono invece riportate nello schema delle operazioni.

Data la seguente porzione di schema:

```
// --------- Circuito
interface Circuito : Elemento_primario
```

```
( extent Circuiti
  keys codice_circuito
)
{
  attribute string tipo_linea;
  attribute string tipo_circuito;
  attribute string tipo_alimentazione;
  attribute range{1,6} tensione;
  attribute range{1,2} frequenza;
  relationship set<Elemento_elettrico> alimenta
       inverse Elemento_elettrico::alimentato;
};


// --------- Elemento_elettrico

interface Elemento_elettrico : Elemento
( extent Elementi_elettrici
)
{
  attribute range{0.0,+inf} corrente_assorbita_per_mille;
  relationship Circuito alimentato
       inverse Circuito::alimenta;
  relationship list<Morsetto> possiede
       inverse Morsetto::fa_parte;
  relationship set<Contatto> contatti
       inverse Contatto::fa_parte;
};

// --------- Elemento_calcolato

interface Elemento_calcolato : Elemento_elettrico
( extent Elementi_calcolati
)
{
  attribute string tipo_elemento_calcolato;
  relationship set<Elemento_primario> deriva
       inverse Elemento_primario::comanda;
};

// --------- Filo
```

```
interface Filo : Elemento_calcolato
( extent Fili
)
{
  attribute range{1,2} tipo_filo;
   // 1 significa 'di fase'
   // 2 significa 'di neutro'
  attribute range{0.0,+inf} sezione;
  attribute range{0.0,+inf} corrente_impiego;
};
```

Si supponga di voler esprimere la proposizione "se un filo fa parte di un circuito di potenza allora la sezione del filo e' dato dalla corrente_impiego/2.5".

Per esprimere una condizione di questo tipo occorre estendere la sintassi delle regole di ODL, in particolare per ammettere la possibilita' di descrivere delle operazioni.
Per la regola in questione, denominata "Fil2", si scrivera' :

```
rule Fil2 forall X in Filo: X.alimentato.tipo_circuito="potenza"
     then X.sezione=
          range{0.0,50.0} dividi(real X.corrente_impiego,real 2.5);
```

dove seguendo la sintassi propria di ODMG-93 per le operazioni, il nome dell'operazione "dividi" viene preceduto dal risultato dell'operazione stessa.

Consideriamo ora la rappresentazione di questa regola in OCDL.
L'antecedente e' una espressione completamente traducibile in OCDL :

Fil2a : $Filo \sqcap \triangle[alimentato: \triangle[\text{tipo\_circuito} : \text{"potenza"}]]$

Del conseguente, invece, riportiamo in OCDL solo il tipo del risultato dell'operazione intersecato ad un nuovo tipo che rappresenta in modo univoco il risultato della specifica funzione "dividi"

Fil2c : $Filo \sqcap \triangle[sezione\_per\_mille : 0.0 \div 50.0 \sqcap Tipo1]$

dove "Tipo1" deve essere un nome di tipo (sia classe che tipo-valore) primitivo senza alcuna descrizione, distinto da tutti gli altri nomi di tipo utilizzati nello schema, il cui scopo è quello di evitare che il sistema confonda un generico tipo "range{0.0,50.0}" dal particolare tipo risultato dell'operazione "dividi" richiamata con particolari parametri attuali.

In base a quanto detto in precedenza sull'univocità degli identificatori di operazioni, possiamo indicare come *Tipo1* l'identificatore $Op1$, e quindi scrivere
Fil2c : $Filo \sqcap \triangle[sezione\_per\_mille : 0.0 \div 50.0 \sqcap Op1]$

A livello formale di modello OCDL, questo comporta il dover considerare anche gli identificatori di operazioni come tipi del modello, e in particolare, come nomi di tipo (sia classe che tipo-valore) primitivi senza alcuna descrizione; si deve quindi ripetere il discorso fatto per i tipi-parametro.
Sinteticamente: un identificatore di operazione viene considerato come un nome di tipo, con

- Interpretazione:

$$\mathcal{I}[Op] \subseteq \mathcal{V}(\mathcal{O})$$

- Schema:

$$\sigma(Op) = \top$$

- Istanza:

$$\mathcal{I}[\sigma(Op)] \subseteq \mathcal{I}[Op]$$

In questo modo, la rule
Fil2 :
$Filo \sqcap \triangle[alimentato: \triangle[\text{tipo\_circuito} : \text{"potenza"}]]$
$\rightarrow$
$Filo \sqcap \triangle[sezione\_per\_mille : 0.0 \div 50.0 \sqcap Op1]$

risulta essere formalmente una rule espressa sul sistema di tipi esteso.
Dall'altra parte, *Op1* appartiene all'insieme $\mathbf{I_M}$ e permette, mediante la funzione $\sigma_{\mathbf{M}}$ di accedere alla quadrupla

(Filo,dividi,[param1:real⊓in,param2:real⊓in],range{0.0,50.0})

che rappresenta l'operazione "dividi".
La dichiarazione di una rule contenente una diversa invocazione della operazione dividi comporta la creazione di un nuovo identificativo.

7.3 Principio di covarianza e controvarianza

In letteratura sono presentati due approcci contrastanti per controllare la consistenza delle interfaccie dei metodi:

1. Un primo approccio applica ai parametri delle operazioni il medesimo principio applicato agli attributi delle classi, ossia il principio di covarianza.

2. Un secondo approccio applica ai parametri delle operazioni il principio opposto a quello applicato agli attributi delle classi, ossia il principio di controvarianza.

Di seguito viene riportata la definizione di sottotipo secondo la teoria dei tipi di dati(E. Bertino e L.D. Martino [EM92]; Bruce and Wegner 1986 [BW86]; Cardelli 1984 [Car84]; Albano et al. 1985 [ACO85]).

Definizione 6 (sottotipo) *Un tipo t e' sottotipo del tipo t' ($t \leq t'$) se:*
(1) le proprieta' di t' sono un sottoinsieme di quelle di t
(2) per ogni operazione m'_t di t' esiste la corrispondente operazione m_t di t tale che:
(a) m_t e m'_t hanno lo stesso nome
(b) m_t e m'_t hanno lo stesso numero di argomenti
(c) l'argomento i di m'_t e' un sottotipo dell'argomento i di m_t **(regola di controvarianza)**
(d) m_t e m'_t restituiscono un valore o entrambi non hanno alcun parametro di ritorno (e) se m_t e m'_t restituiscono un valore allora il tipo del risultato di mt e' un sottotipo del tipo del risultato di m'_t **(regola di covarianza)**

In base a tale definizione è valido il *principio di sostituibilita'*, secondo il quale una istanza di un sottotipo puo' essere sostituita da un supertipo in ogni contesto nel quale quest'ultimo puo' essere legalmente usato.
Altri gruppi di ricerca, come gli autori che hanno sviluppato il sistema O_2 [BDK96], hanno adottato la regola di covarianza anche sui singoli parametri delle operazioni: in questo modo però si possono produrre degli errori in fase di run-time a fronte di una correttezza sintattica.

In Bruce and Wegner 1986 [BW86] si riporta una dimostrazione rigorosa di quanto affermato sopra, sottolineando soprattutto la differenza esistente tra i tipi degli attributi e i tipi dei parametri formali delle operazioni.

Per capire questa affermazione si propone un semplice esempio:

Suppongo di modellare l'aspetto geometrico dei punti del piano cartesiano. I punti sono degli oggetti definiti da due coordinate (x,y), e possono essere positivi oppure negativi.
I punti positivi sono contenuti interamente nel primo quadrante del piano cartesiano, mentre quelli negativi nel terzo quadrante.
Introduco una operazione denominata "disegna" con le due coordinate come parametri di input.

```
interface Punto
 (extent Punti)
{
 attribute range{-1000,1000} x;
 attribute range{-1000,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
};
interface Punto_positivo : Punto
 (extent Punti_positivi)
{
 attribute range{0,1000} x;
 attribute range{0,1000} y;
 void disegna (in range{0,1000} px,in range{0,1000} py);
};
```

In questo esempio ho applicato la regola di covarianza anche sui due parametri dell'operazione (metodo di O_2). Supponiamo di invocare l'operazione

disegna(-10,-10) su un generico oggetto della classe "Punto", se questo oggetto e' anche istanza di "Punto_positivo" allora verra' eseguita l'operazione nella classe piu' specializzata, causando un errore di run-time.
Da un punto di vista pratico questo esempio esclude il principio di covarianza e giustifica solo in parte il principio di controvarianza applicato ai parametri delle funzioni. Per evitare errori in fase di esecuzione delle operazioni, e' sufficiente dichiarare il tipo dei parametri delle sottoclassi **uguale** a quello delle superclassi. Infatti nell'esempio riportato di seguito non vi e' alcun motivo per ampliare il range {0,1000} dei parametri dell'operazione Punto_positivo::disegna.

```
interface Punto
 (extent Punti)
{
 attribute range{-1000,1000} x;
 attribute range{-1000,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
};
interface Punto_positivo : Punto
 (extent Punti_positivi)
{
 attribute range{0,1000} x;
 attribute range{0,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
};
```

In conclusione il principio di controvarianza e' stato adottato per motivi di flessibilita', infatti risulta piuttosto vincolante imporre, nella signature delle operazioni, l'uso degli stessi tipi dichiarati nelle superclassi.

7.3.1 Estensione del controllo nel modello OCDL

Mediante il formalismo adottato nel modello OCDL e presentato nel paragrafo 7.2.1 e' possibile definire il controllo di covarianza o di controvarianza.

1. **Definizione 7 (Schema covariante)** *Uno schema $\sigma_{\mathbf{M}}$ e' covariante sse*
   $\forall$ *coppia di operazioni (C,m,T_s,R),(C',m,T'_s,R') tali che*

(a) $C' \sqsubseteq_{\mathrm{R}} C$,

(b) $R' \sqsubseteq_{\mathrm{R}} R$

(c) T_s *e* T'_s *hanno lo stesso numero di parametri*

è verificato che $T'_s \sqsubseteq_{\mathrm{R}} T_s$

2. **Definizione 8 (Schema controvariante)** *Uno schema* $\sigma_{\mathbf{M}}$ *e' controvariante sse*
∀ coppia di operazioni (C,m,T_s*,R),(C',m,*T'_s*,R') tali che*

(a) $C' \sqsubseteq_{\mathrm{R}} C$,

(b) $R' \sqsubseteq_{\mathrm{R}} R$

(c) T_s *e* T'_s *hanno lo stesso numero di parametri*

è verificato che $T_s \sqsubseteq_{\mathrm{R}} T'_s$

7.4 Il problema del controllo di consistenza di operazioni

In questo paragrafo viene presentato un lavoro teorico, fatto con riferimento al modello O_2 [BDK96], per la rappresentazione del flusso di controllo del codice di una operazione e il conseguente controllo di consistenza.
Nel testo originale [BDK96] si parla di metodi, ma in questa sede si continuera' ad usare il termine "operazioni".

Si considerano due tipi di operazioni:

1. **operazioni base**
   Sono funzioni che resituiscono un valore appartenente ad un tipo specificato. Esse rappresentano gli attributi della classe in uno schema composto solamente da operazioni.

2. **operazioni codificate**
   La codifica delle operazioni avviene rappresentandone il flusso di controllo, in termini di composizione di operazioni, sia base che a loro volta codificate; sono possibili chiamate ricorsive.

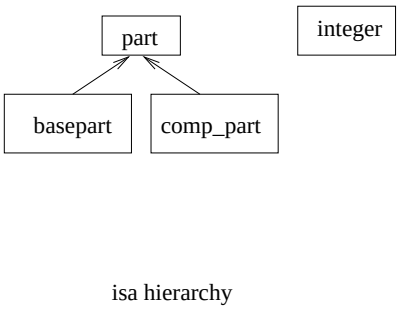

Figura 7.1: Esempio di schema con metodi

In questo modo, si rappresentano i programmi con un paradigma di programmazione basato esclusivamente sulla ereditarieta', l'overloading e il late binding. La semantica di tali operazioni e' una astrazione dell'interprete Smalltalk.

Ad esempio:
In figura 7.1 e' presente una semplice gerarchia di classi in cui e' presente la classe delle parti (Part), le quali possono essere parti di base (basepart) o componenti (comp_part). Lo schema presenta, inoltre le seguenti operazioni base:

cost (part) → integer
assemblingcost (part) → integer

La prima restituisce il costo di una parte, mentre la seconda restituisce il costo di assemblaggio di una parte.
Come esempio di operazione codificata si introduce l'operazione somma (sum). il cui schema e'

sum (integer,integer) → integer

e di questa operazione viene rappresentata la semantica tramite la seguente notazione:

operation sum(assemblingcost(X),cost(X))

in cui "sum" ha come parametri il risultato di due operazioni base: "assem-

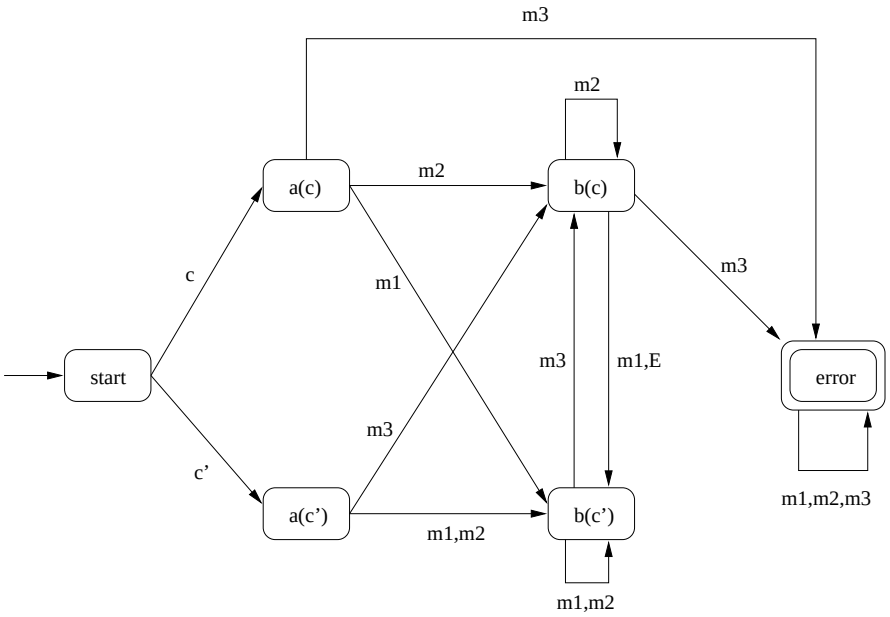

Figura 7.2: Automa a stati finiti per controllare la consistenza

blingcost" e "cost" e X rappresenta il solo parametro di ingresso della operazione. In questo caso l'operazione somma si dice **"monadica"** in quanto prevede un solo parametro.

A questo punto usando una codifica di questo tipo si possono fare verifiche di consistenza sull'intero flusso di controllo dell'operazione.

Nel caso in cui le operazioni dello schema siano monadiche, e' possibile verificare facilmente la consistenza di una singola operazione codificata.

Come esempio occorre considerare uno semplice schema, costituito da due classi in gerarchia **c** e **c'** e una stessa operazione m_2 ridefinita nella sottoclasse c'.

In figura 7.2 si propone un automa a stati finiti in grado di controllare la consistenza di uno schema di operazioni monadico e non ricorsivo.

Lo schema in questione e' rappresentato dalle seguenti operazioni base:

m_1**:c→c'**
m_2**:c→c**
m_2**:c'→c'**
m_3**:c'→c**

E' possibile verificare la consistenza delle seguenti operazioni codificate:

m=$m_1(m_2(m_1(\mathbf{X})))$
m'=$m_3(m_3(m_1(\mathbf{X})))$

La tecnica di controllo di consistenza di queste operazioni deve in sostanza analizzare il flusso di chiamate delle varie operazioni. Tale flusso puo' essere rappresentato come una stringa in cui l'elemento iniziale e' il nome di una classe e gli altri elementi sono i nomi delle operazioni richiamate.
Ad esempio l'operazione m e' rappresentata da due stringhe corrispondenti alle possibili invocazione della operazione: "c$m_1m_2m_1$" e "c'$m_1m_2m_1$".

L'operazione **m** e' consistente: le due stringhe sono rifiutate dall'automa, infatti in entrambi i casi non si raggiunge lo stato finale "error".
L'operazione **m'** e' inconsistente in quanto la stringa "c$m_1m_3m_3$" e' accettata dall'automa. Infatti partendo con il carattere 'c' e proseguendo con il carattere m_1 e m_3 si raggiunge lo stato b(c) dopo il quale si termina nello stato finale con una ulteriore invocazione della operazione m_3.

Nel testo [BDK96] viene dimostrato che, limitatamente al caso di operazioni monadiche, sotto il vincolo di covarianza, il problema della consistenza e' semplificato. Questo non e' verificato per le altre operazioni.

A mio avviso, l'ambito formale qui brevemente illustrato, puo' essere arricchito considerando una gerarchia comprendente anche classi virtuali e operazioni sulle classi virtuali.
Quindi questo controllo puo' essere integrato ed esteso nel modello OCDL. Questo aspetto non e' stato ulteriormente sviluppato nella presente tesi.

Capitolo 8

Realizzazione di estensioni di ODB-Tools

8.1 ODL_Trasl : Il traduttore

8.1.1 Struttura del programma

La sintassi di ODL è stata diffusa in formato *lex&yacc*[1], tale scelta è in accordo con la politica dell'ODMG di rendere ODL facile da implementare.
Attorno alla sintassi Lex&Yacc sono state aggiunte:

1. Le *actions* della parte Yacc. Nelle actions vengono memorizzati tutti i concetti letti dall'ODL, vengono inoltre eseguiti alcuni controlli semantici.

2. Routine di controllo della coerenza dei dati. Tra queste routine è presente l'algoritmo di traduzione delle rules.

3. Routine di stampa in formato OCDL.

Il programma è composto dai seguenti moduli

- modulo principale (`c_main.c`):
  Si ha l'inizializzazione delle variabili globali, e chiama in sequenza: *il parser*, *il controllore coerenza*, *la routine di stampa dei concetti in OCDL*

- parser della sintassi ODL:
  È composto dal modulo *lex* (`odl.l`) e dal modulo *yacc* (`odl.y`). Svolge

[1] si veda appendice D

il controllo sintattico della sintassi ODL grazie alle routine generate con Lex&Yacc. Durante tale controllo riempie le strutture dati in memoria.

- controllore di coerenza (`c_coeren.c`):
  Controlla l'univocità dei nomi de: `const`, `struct`, `typedef`, `interfaces`, ed `operation`. Controlla che esistano i campi inversi delle "relazioni" e calcola la traduzione OCDL delle rules.

- routine di stampa dei concetti in OCDL (`c_procdl.c`):
  Dalle strutture dati allocate dal parser estrae la forma OCDL.

Di seguito saranno descritte solo le parti del programma interessate alla estensione delle operazioni. Per eventuali chiarimenti sulla struttura originale del traduttore si veda la Tesi di laurea di A. Corni [Cor97].

8.1.2 Estensione della grammatica di ODL

Per permettere l'utilizzo delle operazioni, si e' estesa la sintassi ODL, quindi si e' modificato il parser di lex&yacc (in paricolare il file odl.y).
Per quanto concerne la dichiarazione delle operazioni all'interno delle interface delle classi, si e' utilizzata la sintassi standard ODMG-93.
Per la dichiarazione delle operazioni all'interno delle rule, invece, si è cercata una sintassi che fosse:

- Tale da rendere intuitivo il significato delle regole
- Coerente con le sintassi proposte da ODMG.

Di seguito si riporta la parte della sintassi ODL che riguarda il contesto nel quale si puo' dichiarare una operazione in una interface di una classe:

$$
\begin{array}{rcl}
\langle \text{OpDcl} \rangle & ::= & \langle \text{OpTypeSpec} \rangle \ \langle \text{Identifier} \rangle \ \langle \text{ParameterDcls} \rangle \\
\langle \text{ParameterDcls} \rangle & ::= & \textbf{(} \ \langle \text{ParamDclList} \rangle \ \textbf{)} \ \mid \ \textbf{( )} \\
\langle \text{ParamDclList} \rangle & ::= & \langle \text{ParamDcl} \rangle \ \mid \\
 & & \langle \text{ParamDcl} \rangle \ \textbf{,} \ \langle \text{ParamDclList} \rangle \\
\langle \text{ParamDcl} \rangle & ::= & [\langle \text{ParamAttribute} \rangle] \ \langle \text{SimpleTypeSpec} \rangle \ \langle \text{Identifier} \rangle \\
\langle \text{ParamAttribute} \rangle & ::= & \textbf{IN} \ \mid \ \textbf{OUT} \ \mid \ \textbf{INOUT}
\end{array}
$$

Di seguito si riporta la parte della sintassi ODL che riguarda il contesto nel quale si puo' dichiarare una operazione in una rule:

< RuleDcl > ::= **rule** < Identifier >
< RuleAntecedente > **then** < RuleConseguente >

< RuleAntecedente > ::= < Forall > < Identifier > **in** < Identifier > **:**
< RuleBodylist >

< RuleConseguente > ::= < RuleBodyList >

< RuleBodyList > ::= **(** < RuleBodyList > **)** |
< RuleBody > |
< RuleBodylist > **and** < RuleBody > |
< RuleBodylist > **and (** < RuleBodyList > **)**

< RuleBody > ::= < DottedName > < RuleConstOp > < LiteralValue > |
< DottedName > < RuleConstOp > < RuleCast >
< LiteralValue > |
< DottedName > **in** < SimpleTypeSpec > |
< ForAll > < Identifier > **in** < DottedName > **:**
< RuleBodylist > |
exists < Identifier > **in** < DottedName > **:**
< RuleBodylist > |
< DottedName > **=**
< SimpleTypeSpec > < FunctionDef >

< RuleConstOp > ::= = | >= | <= | < | >

< RuleCast > ::= **(** < SimpleTypeSpec > **)**

< DottedName > ::= < Identifier > |
< Identifier > **.** < DottedName >

< ForAll > ::= **for all** | **forall**

```
< FunctionDef >          ::= < Identifier > ( < DottedNameList > )
< DottedNameList >       ::= [< SimpleTypeSpec >] < DottedName > |
                             [< SimpleTypeSpec >] < LiteralValue > |
                             [< SimpleTypeSpec >] < DottedName > ,
                             < DottedNameList > |
                             [< SimpleTypeSpec >] < LiteralValue > ,
                             < DottedNameList >
```

Osservazioni sulla grammatica Osservando la sintassi di ODL-esteso e' possibile fare alcune considerazioni interessanti:

- L'attributo del parametro (ParamAttribute), premesso al tipo del parametro e' facoltativo. Nel caso non venga indicato viene considerato IN.
- Le operazioni dichiarate nel corpo delle rule possono appartenere solamente ad un sottoinsieme delle operazioni. Esse sono considerate delle funzioni matematiche. Questo significa che sono vincolate a rispettare le seguenti regole:
  - devono restituire un risultato
  - devono possedere almeno un parametro
  - ogni parametro deve avere attributo IN. Infatti non e' permesso indicare l'attributo del parametro nel caso si tratti di funzioni.

8.1.3 Le strutture dati

Di seguito viene presentata a grandi linee la struttura con cui è memorizzato uno schema, inoltre è presente una accurata descrizione delle strutture che conservano informazioni relative alle operazioni.

Rappresentazione di un'*interface* Vediamo come sono memorizzate le informazioni di un'*interface*.

```
struct s_interface_type
   {
```

```
char         *name;
             /* nome interfaccia */
char         fg_interf_view;
             /* indica se l'"interfaccia" e' un'interfaccia
              * oppure una view
              * vale:
              *   'i'    interfaccia
              *   'v'   view
              */
struct       s_iner_list  *iner;
             /* lista delle superclassi (eriditarieta')*/
struct       s_prop_list  *prop;
             /* lista delle proprieta' */
struct s_interface_type *next;
             /* lista globale delle interfaces */
char         fg_used;
             /* flag usato per evitare i cicli
              * nella ricerca ricorsiva degli attributi
              * nelle classi e nei suoi genitori
              * puo' valere:
              *   ' ' se non usato
              *   '*' se usata
              */
}
```

Il campo `prop` punta alla lista delle proprietà.
Il campo `next` serve per la gestione della lista globale di tutte le `interface` dello schema.

Rappresentazione delle *proprieta'* Vediomo come vengono memorizzate le propieta' di una classe.

```
struct s_prop_list
   {
   char prop_type;
                /*
                 * prop_type: puo' valere:
                 *
                 * a attributo
                 *   punta ad una struttura s_attr_type
                 * r relazione
```

```
                    *   punta ad una struttura s_rela_type
                    * t tipo di dato
                    * c costanti
                    * o operations  operazioni
                    * --- tipi previsti ma IGNORATI:
                    * e exceptions  gestione eccezioni
                    */
            union       /*punta alla struttura dati appropriata*/
               {
               struct s_attr_type   *a;
               struct s_rela_type   *r;
               struct s_type_list   *t;
               struct s_const_type  *c;
               struct s_operation_param  *o;
               } prop;
            struct s_prop_list *next;
            }
            *Prop_list;
```

Un record della lista delle proprietà di un'`interface` può descrivere:

- a attributo, punta ad una struttura s_attr_type
- r relazione, punta ad una struttura s_rela_type
- t tipo di dato (`typedef`)
- c costanti
- o operazione, punta ad una struttura s_operation_param

In funzione del tipo di proprieta' e' possibile accedere alle informazioni specifiche mediante un apposito puntatore.

Rappresentazione delle *operazioni* Le operazioni contenute nello schema del database vengono memorizzate nella lista delle proprieta' associate ad ogni classe. Il record che rappresenta l'operazione contiene il puntatore alla lista dei parametri. Tale lista e' una sequenza di strutture s_operation_param. Vediamo tale struttura:

```
struct s_operation_param
{
 char   type;         /* tipo di parametro */
 union
 {
  char   *OperationName;
```

```
  char   *DottedName;
  char   *Value;
 } r;
 char *attribute_param;// in o out o inout
 char *param_type; //se 'n' e' il tipo di ritorno
 struct s_declarator_type *ParamDeclarator;/* e' il "nome" */
 struct s_operation_param *next;
} *Operation_param;
```

La struttura ha un duplice utilizzo:

- permette di memorizzare le operazioni dichiarate nella interface delle classi
- permette di memorizzare le funzioni dichiarate nelle rule

Descrizione dei campi della struttura :

- **type** indica il tipo di parametro
  - se type e' uguale a 'n' allora si e' nel primo elemento della lista dei parametri, quindi si memorizza:
    - il nome della operazione in **OperationName**
    - il tipo di ritorno in **paramtype**
  - se type e' uguale a 'p' allora si e' dichiarata l'operazione all'interno di una interface di una classe, in tal caso si memorizza:
    - il nome del parametro in **ParamDeclarator**
    - l'attributo del parametro in **attribute_param**
    - il tipo del parametro **param_type**
  - se type e' uguale a 'd' si e' dichiarato l'operazione all'interno di una rule ed il parametro presenta un DottedName come valore di passaggio.In tal caso si memorizza:

    * l'attributo del parametro in **attribute_param**

    * il nome del dottedname in **DottedName**

    * il tipo del parametro in **param_type**. Il tipo del parametro e' facoltativo, nel caso in cui non venga specificato esso viene calcolato dal programma.
    * il nome del parametro in **ParamDeclarator**. Nel caso in cui l'operazione venga dichiarata in una rule, nella sintassi ODL non appare il nome del parametro formale, ma solamente il nome del parametro attuale, il quale puo' essere un dottedname. In questo caso il programma assegna un nome al parametro formale e lo assegna alla variabile ParamDeclarator. Tale nome viene generato in modo molto semplice, inizia con "param" seguito da un numero progressivo che identifica ogni parametro (ad esempio param1, param2 ecc...)

  – se type e' uguale a 'v' si e' dichiarata l'operazione all'interno di una rule ed il parametro presenta una costante come valore di passaggio.In tal caso si memorizza:

    * l'attributo del parametro in **attribute_param**

    * il valore in **Value**

    * il tipo del parametro in **param_type**. Anche in questo caso il tipo puo' essere ricavato dal programma.
    * il nome del parametro in **ParamDeclarator**. Anche in questo caso il nome del parametro viene determinato dal programma.

- **next** il puntatore al prossimo elemento della lista

Rappresentazione delle *rule* Vediamo in che modo le informazioni relative alle rules sono memorizzate, ovvero la rappresentazione interna delle informazioni delle rules lette in fase di parsing dal formato ODL-esteso.

```
struct s_rule_type
{
   char   *name;          /* nome della rule */
```

```
   char   *ocdlante;     /* output OCDL della relazione */
   char   *ocdlcons;     /* questo campo \'e "riempito" */
                         /* in fase di validazione ed usato */
                         /* in fase di stampa dell'OCDL */
   struct s_rule_body_list *ante;
   struct s_rule_body_list *cons;
   /* lista globale di tutte le regole */
   struct s_rule_type      *next;
}
```

La struct `s_rule_type` serve per memorizzare le varie rules. Tutte le rules dichiarate in uno schema sono unite in una lista. Come si vede, una `rule` è descritta da una parte antecedente ed una conseguente. È previsto che in questa struttura sia memorizzata la traduzione della rule in formato OCDL (nei campi `ocdlante` e `ocdlcons`). Le due condizioni antecedenti e conseguenti sono descritte come una lista di condizioni.

Di seguito è riportata la struttura dati che permette di memorizzare una lista di condizioni. Si può notare che un record di tale lista è in grado di descrivere uno qualunque dei costrutti base che compongono una condizione di una regola.

```
struct s_rule_body_list
{
   char type;
   char fg_ok;
    /* variabile di comodo per sapere se una data condizione
     * e' gia' stata considerata
     * puo' valere:
     *   ' '  condizione NON ancora considerata
     *   '*'  condizione gia' considerata
     */
   char fg_ok1;
    /* variabile di comodo
     * usata SOLO per la body_list del primo livello
     * della parte conseguente di una rule
     * serve per sapere se una data condizione
     * e' gia' stata considerata
     * infatti nel caso particolare del primo livello
     * di una condizione conseguente
     * si hanno due tipi di condzioni
     *  1. quelle che coinvolgono X come iteratore
```

```
  *     es:   X.int_value = 10
  *     queste devono essere racchiuse tra par. quadre
  *  2. quelle che coinvolgono X in quanto indica
  *     il membro dell'estensione
  *     es:   X in TManager
  *     queste devono essere messe in and con il tipo classe
  *     es:  Manager & TManager ...
  * pu\'o valere:
  *   ' '  condizione NON ancora considerata
  *        questo \'e il valore di default
  *   '*'  condizione gi\'a considerata
  */
char *dottedname;           /* nome variabile interessata */
union
{
      struct
      {     /* in questo caso
             * dottedname e' la variabile da mettere
             * in relazione con la costante
             */
            char *operator;
            char *cast;        /* NULL se manca il cast */
            char *value;
      } c;
      struct
      {     /* in questo caso
             * dottedname e' la variabile su cui imporre
             * il tipo
             */
            char *type;         /* identif. tipo  */
    /* puntatore alla operazione */
            struct s_operation_param *param_list;
      } i;
      struct
      {     /* in questo caso
             * dottedname \'e la lista su cui iterare
             */
            char fg_forall;    /*  puo valere:
                                *   'f'  forall
                                *   'e'  exists
                                *        significa che il
                                *        tipo \'e un'exists
                                * questo flag \'e stato
```

```
                                    * introdotto in quanto i
                                    * tipi EXISTS e FORALL
                                    * hanno quasi la stessa traduzione
                                    * in comune
                                    */
                char *iterator;    /* nome iteratore */
                struct s_rule_body_list *body;
          } f;
   } r;
   struct s_rule_body_list *next;
}
```

Descrizione della struttura :

- **type** indica il tipo di parametro, il quale puo' valere:
  - 'c' dichiarazione di costante
  - 'f' la regola è un forall o un'exists
  - 'i' dichiarazione di tipo
    In questo caso particolare, puo' essere inserita nella condizione una dichiarazione di operazione. Infatti nella struttura **i** e' presente un campo, denominato **param_list**, che punta alla lista dei parametri di una operazione (**s_operation_param**).

Quando una operazione e' dichiarata all'interno di una rule, la sua lista dei parametri viene accodata alle proprieta' della classe alla quale si riferisce. Vengono memorizzati i parametri di passaggio (Dottedname oppure valori costanti) e vengono creati dal programma i nomi dei parametri formali da inserire nella signature della operazione stessa.

8.1.4 Descrizione delle funzioni

Utilizzando il parser generato da Lex&Yacc e' possibile riempire le strutture dati in memoria con le informazioni relative allo schema presentato in input. Di seguito si presentano le funzioni che leggono queste strutture dati per eseguire alcuni controlli di consistenza e restituire in output il file in formato OCDL.

Funzioni di gestione delle rule Dal diagramma di figura 8.1 è possibile vedere come è strutturato l'algoritmo di controllo-semantico e traduzione delle `rule` secondo la metodologia PHOS.

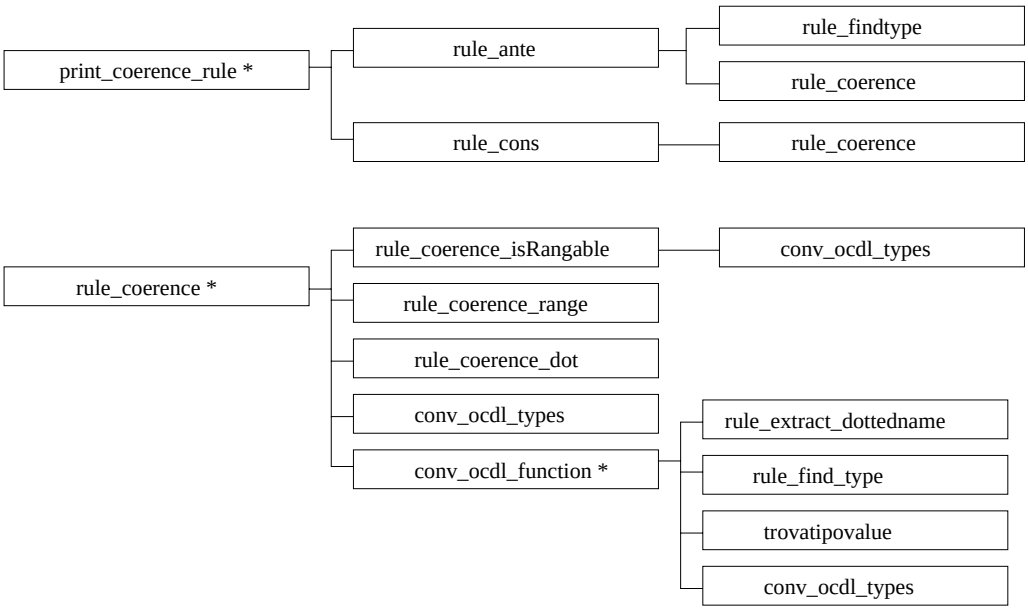

Figura 8.1: Legami tra le funzioni di gestione delle `rule`

Di seguito sono presentate le funzioni che realizzano l'algoritmo. Particolare attenzione e' posta nel descrivere le funzioni che trattano le operazioni.

- **rule_coerence**
  La funzione `rule_coerence` effettua il controllo e la traduzione in OCDL di una qualunque lista di *condizioni*.
  input Una lista di condizioni, cioè una lista di record definiti dalla struct
  `s_rule_body_list`.
  output Una stringa in cui le singole condizioni sono tradotte in OCDL e separate tra loro da virgole.
  descrizione Questa funzione effettua la gestione ricorsiva del corpo delle rules cioè di una lista di condizioni, inoltre essa traduce ogni singola condizione della lista in formalismo ocdl.
  Quando si incontra un predicato contenente una operazione, viene richiamata la funzione **conv_ocdl_function** e viene passata ad essa il puntatore alla lista dei parametri.

- **conv_ocdl_function**
  La funzione `conv_ocdl_function` effettua il controllo e la traduzione

in OCDL di una *operazione* dichiarata all'interno della rule.
input Una lista di parametri, il primo dei quali e' solo fittizio e contiene il nome della operazione.
output Una stringa contenente la traduzione in OCDL della operazione.
descrizione Questa funzione itera sulla lista dei parametri e traduce ognuno di essi in formalismo ocdl. Per fare cio' utilizza un insieme di funzioni gia' presenti in ODL_trasl. Ad esempio:

- **conv_ocdl_types**
  Converte i singoli tipi ODL-ODMG in tipi OCDL.
- **trovatipovalue**
  Nel caso il parametro di passaggio utilizzato e' una costante, questa funzione e' in grado di restituire il tipo di tale costante. Questa funzione permette di rendere facoltativa la dichiarazione del tipo di un valore costante.
- **rule_extract_dottedname**
  Estrae l'attributo indicato da un dottedname.
- **rule_findtype**
  Restituisce il tipo di un dotted-name presentato in input. Questa funzione permette di rendere falcotativa la dichiarazione del tipo di un dottedname.

Funzioni di gestione delle operazioni Dopo aver effettuato il controllo delle rule, e' necessario accodare le operazioni dichiarate in esse all'elenco generale delle operazioni dello schema. Infine e' necessario convertire tutte le operazioni in formalismo OCDL.
Dal diagramma PHOS di figura 8.2 è possibile vedere come sono legate le funzioni che si occupano di questi compiti.

- **print_ocdl**
  Scrive l'intero schema in formato OCDL nel file di estensione `sc`
- **print_ocdl_all_operation**
  Scrive l'intera lista delle operazioni nel file di estensione `sc`. Prima di effettuare l'iterazione sulle operazioni viene richiamata la funzione **aggiorna_operazioni**, la quale accoda le operazioni dichiarate nelle rule a quelle dichiarate nelle interface delle classi. In questo modo

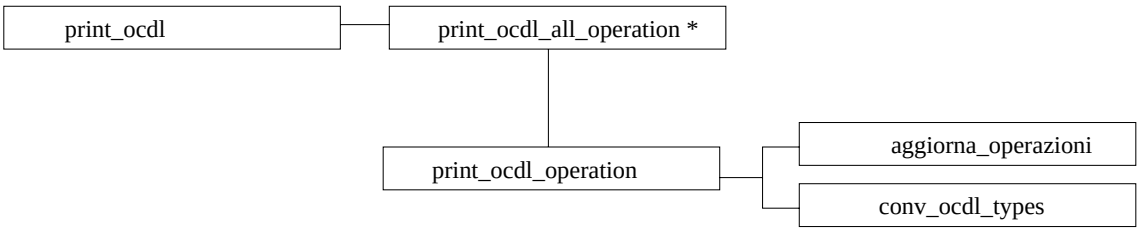

Figura 8.2: Legami tra le funzioni di gestione delle `operazioni`

opera sulla globalita' delle operazioni.
Per ogni operazione dello schema viene richiamata la funzione **print_ocdl_operation**, la quale si occupa di una singola operazione.

8.1.5 Esempio

Considero l'esempio precedente che si riporta di seguito per comodita' del lettore:

```
// --------- Filo

interface Filo : Elemento_calcolato
( extent Fili
)
{
  attribute range{1,2} tipo_filo;
   // 1 significa 'di fase'
   // 2 significa 'di neutro'
  attribute range{0.0,+inf} sezione;
//corrente che passa per il filo
  attribute range{0.0,+inf} corrente_impiego;
};

//se il circuito e' di potenza la sezione e' la corrente_impiego/2.5
rule Fil2 forall X in Filo: X.alimentato.tipo_circuito="potenza"
     then X.sezione=
          range{0.0,50.0} dividi(real X.corrente_impiego,real 2.5);
```

In figura 8.3 si illustra come viene memorizzata questa regola. Sono presenti la parte antecedente e la parte conseguente della regola, nella parte

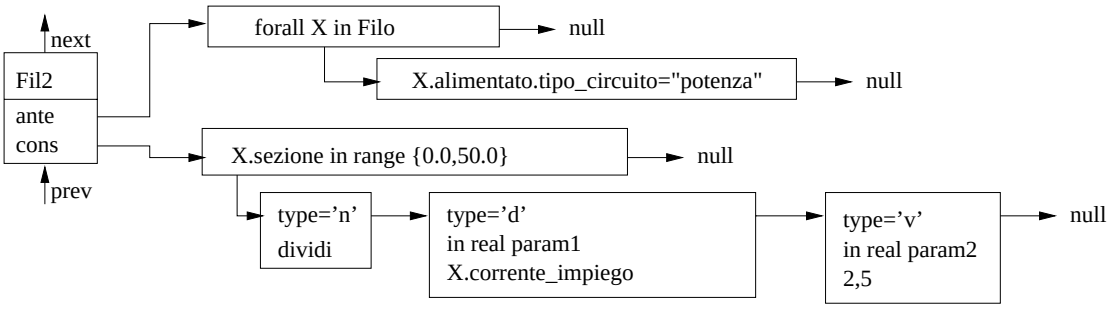

Figura 8.3: Struttura in memoria della rule Fil2

conseguente e' presente una operazione, quindi un puntatore a una lista di parametri.

8.2 OCDL_Designer

8.2.1 Le struttura dati

Lo schema viene memorizzato in tre strutture a lista:

- **listaN** per i tipi valore e i tipi classe
- **listaB** per i tipi base
- **listaO** per le operazioni

L'estensione del formalismo alle operazioni interessa solo due liste:

1. ListaO che contiene tutte le operazioni dello schema
2. ListaN che contiene le rule, all'interno delle quali si possono trovare dei predicati contenenti operazioni.

La lista ListaN Gli elementi della lista ListaN hanno la seguente struttura:

```
typedef struct lN{
char *name; /*nome di tipo o di classe */
int type; /*costante che identifica il tipo */
int new; /*uguale a TRUE se e' un nuovo nome */
```

```
L_sigma *sigma; /*puntatore alla descrizione iniziale*/
L_sigma *iota;  /*puntatore alle descrizione trasformata*/
L_gs *gs; /*puntatore alla lista dei gs */
struct  lN *next;
} lN;
```

I campi, che interessano il lavoro di questa tesi, hanno i seguenti significati:

- name : stringa che descrive il nome di tipo o di classe
- type : costante che identifica il tipo, puo' assumere i seguenti valori:
  - T : tipo valore
  - C : classe primitiva
  - D : classe virtuale
  - RA_V o RA_T: antecedente di una regola
  - RC_V o RC_T: conseguente di una regola
  - OP : tipo operazione
- sigma : puntatore alla lista che rappresenta la descrizione originale

Gli elementi della lista sigma hanno la seguente struttura:

```
typedef struct L_sigma{
char *name;       /* nomi di tipi o di classi */
int type;       /* costante che identifica il tipo */
void *field;       /*puntatore per informazioni aggiuntive*/
struct L_sigma *next; /*punt. per rappresentare descrizioni annidate*/
struct L_sigma *and;  /*punt. per rappresentare congiunzioni */
char *attribute;      /*per conservare (in/out/inout) */
} L_sigma;
```

I campi hanno i seguenti significati:

- name : stringa che descrive il nome di tipo o di classe
- type : costante che identifica il tipo, puo' assumere i seguenti valori:
  - B : tipo base
  - T : tipo valore
  - C : classe primitiva

  - D : classe virtuale
  - RA_V o RA_T: antecedente di una regola
  - RC_V o RC_T: conseguente di una regola
  - SET : insieme
  - SEQ : sequenza
  - EN : ennupla
  - ITEM : attributo di una ennupla
  - DELTA : tipo oggetto
  - OP : tipo operazione

- field : puntatore a elementi di strutture contenenti informazioni aggiuntive
- and : puntatore ad un elemento L_sigma utilizzato per rappresentare le intersezioni nelle descrizioni

La lista ListaO Questa lista ha una struttura perfettamente identita alla lista ListaN, ma contiene esclusivamente le operazioni dello schema. Quindi gli elementi di ListaO sono strutture di tipo **lN** che contengono nel campo type il valore **OP**.

Ulteriori informazioni relative alla struttura del modulo OCDL-Designer si trovano in [Gar95]

8.2.2 Esempio

In questo paragrafo si intende evidenziare come la rule Fil2 viene memorizzata all'interno di OCDL-Designer.
Considero l'esempio precedente che si riporta di seguito per comodita' del lettore:

```
// --------- Filo

interface Filo : Elemento_calcolato
( extent Fili
)
```

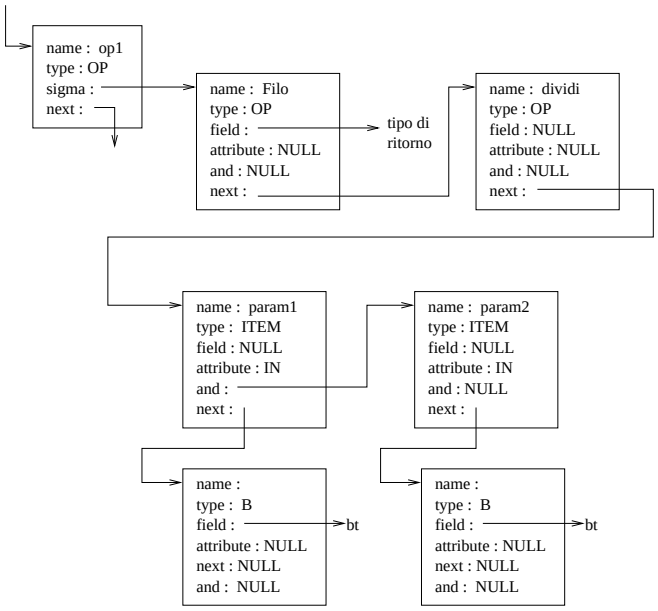

Figura 8.4: L'operazione "dividi" nella lista delle operazioni

```
{
  attribute range{1,2} tipo_filo;
   // 1 significa 'di fase'
   // 2 significa 'di neutro'
  attribute range{0.0,+inf} sezione;
//corrente che passa per il filo
  attribute range{0.0,+inf} corrente_impiego;
};

//se il circuito e' di potenza la sezione e' la corrente\_impiego/2.5\\
rule Fil2 forall X in Filo: X.alimentato.tipo_circuito="potenza"
     then X.sezione=
          range{0.0,50.0} dividi(real X.corrente_impiego,real 2.5);
```

L'operazione "dividi" viene memorizzata due volte all'interno di OCDL-Designer. Nella lista delle operazioni dello schema (ListaO) e all'interno della rule Fil2.
In figura 8.4 si illustra la lista delle operazioni dello schema.
In figura 8.5 si illustra la medesima operazione all'interno della rule.

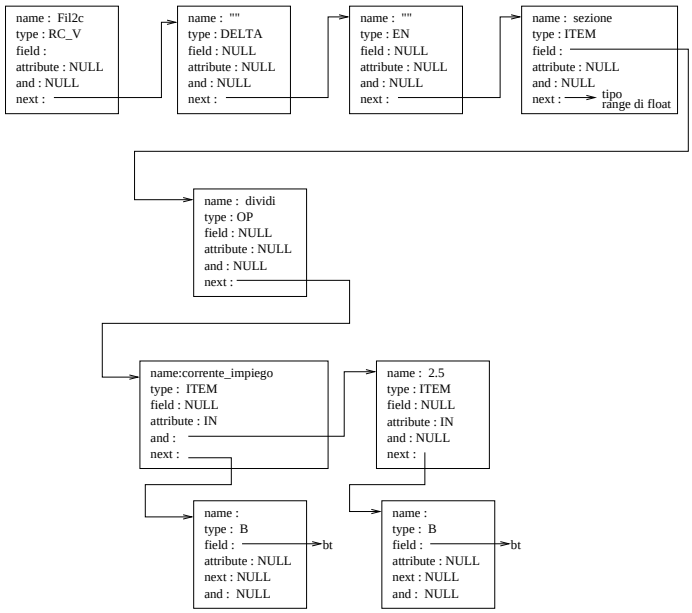

Figura 8.5: La classa virtuale Fil2c

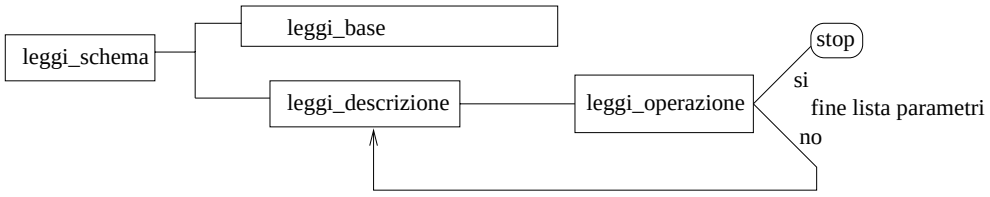

Figura 8.6: Legami tra le funzioni di `OCDL-Designer`

8.2.3 Descrizione delle funzioni

In figura 8.6 si presenta il diagramma PHOS della parte del programma OCDL-Designer che si occupa della lettura dello schema in formalismo ocdl.

Funzioni per la lettura di OCDL

- **leggi_schema**
  Funzione che legge il file ocdl contenente lo schema. Essa esegue un ciclo fino a che non incontra il carattere ".", ad ogni passo legge una definizione. Una definizione puo' essere:

  – un tipo base, in tal caso viene creato un elemento di tipo **bt** e inserito in coda alla lista **listaB**, dopodiche' viene chiamata la

funzione **leggi_base** che legge la descrizione relativa.

- un tipo valore o di una classe, in tal caso viene creato un elemento di tipo **lN** e inserito in coda alla lista **listaN**, dopodiche' viene chiamata la funzione **leggi_descrizione** che legge la descrizione relativa al nome definito.
- una operazione, in tal caso viene creato un elemento di tipo **lN** e inserito in coda alla lista **listaO**, dopodiche' viene chiamata la funzione **leggi_operazione** che legge la descrizione relativa alla operazione definita.

- **leggi_descrizione**
  E' una funzione ricorsiva utilizzata per la lettura della descrizione di un tipo valore o di una classe. Ad ogni chiamata della funzione viene creato un elemento di tipo **L_sigma**, la ricorsione ha termine quando il tipo letto e' un tipo base o un nome.

- **leggi_operazione**
  Questa funzione legge la dichiarazione di una operazione. Essa e' in grado di leggere il nome della operazione, dopodiche' richiama iterativamente la funzione **leggi_descrizione** per ogni parametro. In questo modo si e' ottenuta la massima riusabilita' del software. La funzione **leggi_operazione** viene richiamata in due punti del programma:
  - iterativamente per ogni operazione presenta nella lista globale **listaO**.
  - ogni volta in cui sia presente la dichiarazione di una operazione all'interno di una rule.

Funzioni per la scrittura della forma canonica Per quanto concerne la scrittura del contenuto delle strutture dati nella forma canonica, rappresentata dal file di estensione **fc**, l'algoritmo relativo alle operazioni e' contenuto nella funzione **scrivDesc**.
La funzione **scrivDesc** e' una funzione ricorsiva in grado di scrivere qualsiasi descrizione in ocdl nel file **fc**.

- Nel caso essa incontri la dichiarazione di una operazione all'interno di una rule, richiama se stessa sulla lista dei parametri per generare in output l'intera descrizione ocdl della operazione stessa.

- Per quanto riguarda la scrittura di tutte le operazioni, la funzione **scrivi_operazioni** scorre la lista **listaO** e richiama, per ogni operazione, la funzione scriviDesc, la quale si occupa della scrittura nel file **fc** della signature della operazione.

In questo caso e' necessario precisare che la scrittura nella forma canonica e' limitata alla copiatura del formalismo ocdl, in quanto non viene fatto nessun controllo di consistenza sulla signature delle operazioni.

Capitolo 9

Risultati e note conclusive

9.1 Schema concettuale del dominio Elet con regole espresse in ODB-Tools

In figura 9.1 e' riportata la rappresentazione grafica dello schema del database ottenuto con l'applet java **scvisual** di ODB-Tools.
In questo schema, a differenza di quello presentato nel capitolo 4, sono presenti anche le rules. Osservando lo schema del database e' possibile fare alcune considerazioni:

- ODB_Tools e' in grado di mostrare relazioni tra classi non esplicitamente evidenti. Ad esempio osservando la rule "Att1a", e' possibile notare, tra le altre, relazioni con le classi: "Impianto", "Contatto" e "Morsetto". Questo avviene poiche' la rule "Att1a" eredita tutti gli attributi e le relazioni delle sue superclassi.

- ODB_Tools e' in grado di rilevare le classi equivalenti. Esse si possono individuare facilmente, in quanto sono rappresentate nello stesso rettangolo nello schema grafico.
  Ad esempio cio' avviene per la classe primitiva "Sottoelemento" e per la classe virtuale "Sott5c". Poiche' "Sott5c" e' il conseguente di una regola, questa equivalenza dimostra la ridondanza di "Sott5c" e la sua sostituibilita' con la classe "Sottoelemento"

  Questo significa che tali rule non arricchiscono la semantica dello schema e quindi sono superflue.

- E' interessante osservare come sono state raggruppate le rule Sott2, Sott4, Sott6 e Sott8. Esse hanno la parte conseguente identica, cio' e'

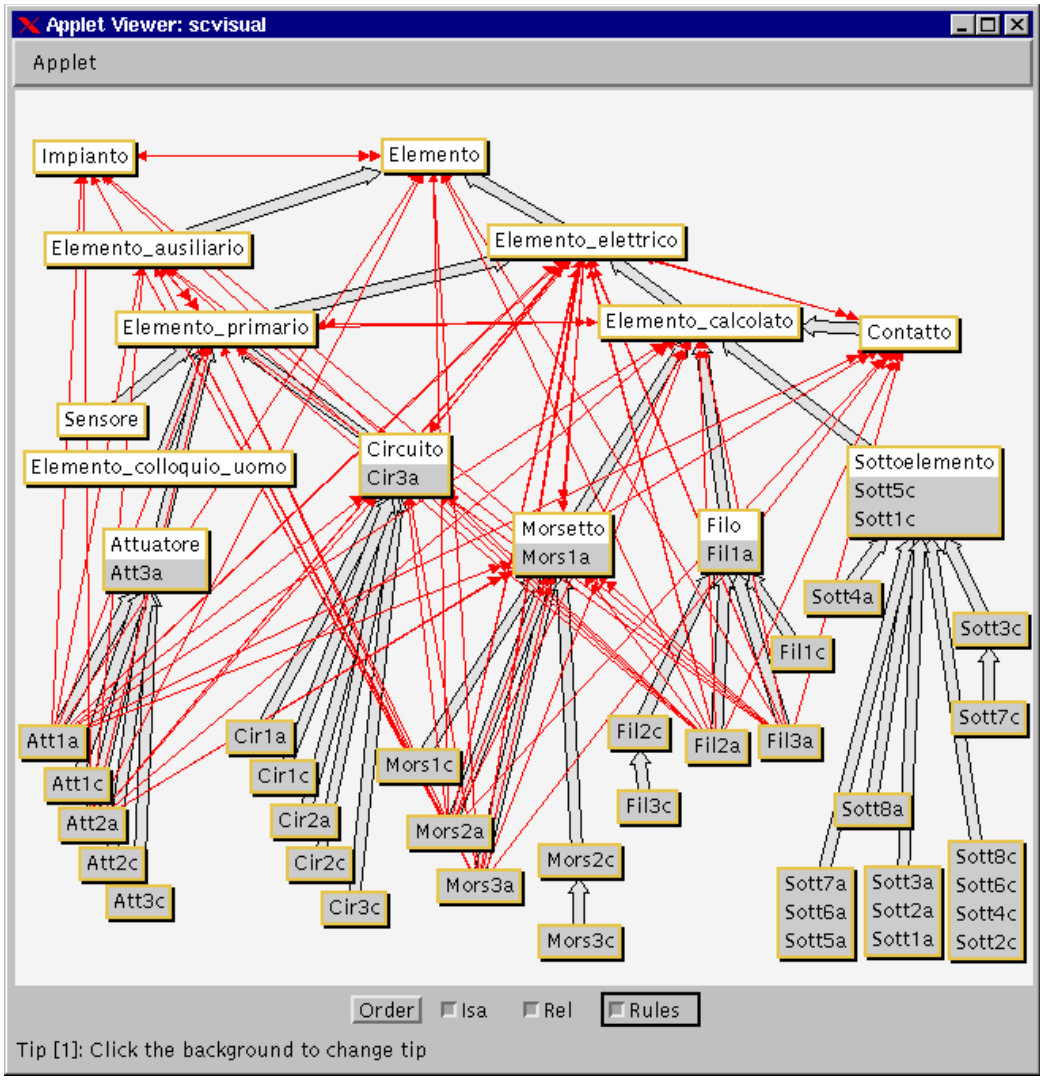

Figura 9.1: Schema del database con regole

dovuto alla mancanza dell'operatore logico OR. Quindi si e' reso necessario separare le clausole in OR in diverse rule. Il sistema ODB_Tools e' stato in grado comunque di accorpare le classi sottoposte a tale vincolo.

- E' interessante il caso in cui una rule eredita da un'altra rule, come ad esempio "Mors3c" e "Mors2c". In questo caso gli oggetti che soddisfano la rule "Mors3c" soddisfano anche la rule Mors2c.

Introducendo l'operatore logico OR e' possibile semplificare notevolmente lo schema di figura 9.1. Le rule che contengono l'unione logica di predicati nella parte antecedente possono essere rappresentate:

- mediante una sola classe virtuale corrispondente alla parte conseguente, si sceglie un nome rappresentativo e gli altri nomi sono considerati alias.
- mediante un nuovo simbolo grafico che rappresenta l'insieme di classi virtuali corrispondenti ai diversi antecedenti uniti in OR.

Di seguito si riportano due rule aventi la parte conseguente coincidente:

```
rule Sott1 forall X in Sottoelemento: X.tipo_elemento_calcolato="Q"
     then X.categoria_impiego=string categoria_impiego(X.deriva);

rule Sott5 forall X in Sottoelemento: X.tipo_elemento_calcolato="K"
     then X.categoria_impiego=string categoria_impiego(X.deriva);
```

Le rule "Sott1" e "Sott5" possono essere sostituite dalla seguente rule, contenente l'operatore logico OR:

```
rule Sott_OR forall X in Sottoelemento:
     X.tipo_elemento_calcolato="Q" OR X.tipo_elemento_calcolato="K"
     then X.categoria_impiego=string categoria_impiego(X.deriva);
```

9.2 Note conclusive

La presente tesi e' nata da una esigenza pratica, offerta dalla ditta A.S.T. System Automation, riguardante la modellazione di un impianto elettrico industriale. Tale problematica ha permesso :

- di realizzare un prototipo software mirato alle esigenze specifiche del progettista di impianti elettrici

- di realizzare un componente riutilizzabile per rendere persistenti gli oggetti di Visual C++
- Eseguire un test di rilevante importanza al prototipo ODB_Tools
- ad ODB_Tools di accettare schemi dotati di operazioni
- al progettista di basi di dati che utilizza ODB_Tools di esprimere i vincoli di integrita' in maniera piu' semplice ed intuitiva utilizzando predicati contenenti operazioni.
- di approfondire le problematiche relative al controllo di consistenza sulle operazioni di uno schema di basi di dati orientate agli oggetti. Questo ultimo aspetto, risulta essere il piu' importante per il futuro di ODB_Tools. Infatti e' necessario estendere il codice di OCDL-Designer al fine di eseguire un controllo sui parametri delle operazioni, sia che esso segua il principio di covarianza o il principio di controvarianza.

Appendice A

Schema del dominio Elet

A.1 Schema ODL-ODMG93

```
/* ********************************************************** */
/* ***** Struttura di un impianto elettrico industriale,     */
/* ****** espressa mediante ODL                               */
/* ******                                                     */
/* ****** realizzato da Stefano Riccio                        */
/* ********************************************************** */

/* ---------------------------------------------------------- */
// --------- Impianto
// * classe degli impianti elettrici realizzati
/* ---------------------------------------------------------- */
interface Impianto
( extent Impianti
  keys codice_impianto
)
{
  attribute string codice_impianto;
  attribute string descrizione_impianto;
  attribute string luogo;
  attribute string codice_cliente;
  attribute integer temperatura_minima;
  attribute integer temperatura_massima;
  attribute range{0,100} umidita_minima;
  attribute range{0,100} umidita_massima;
  attribute integer altitudine;
```

```
  relationship set<Elemento> composto
       inverse Elemento::impianto;
};

/* ------------------------------------------------------------ */
// --------- Elemento
// * generico elemento di un impianto
/* ------------------------------------------------------------ */
interface Elemento
( extent Elementi
  keys codice_elemento
)
{
  attribute string codice_elemento;
  attribute string casa_costruttrice;
  attribute string serie;
  attribute string modello;
  attribute string codice_commerciale;
  attribute string descrizione_elemento;
  attribute string codice_principale_sigla;
  attribute string materiale_sigla;
  relationship Impianto impianto
       inverse Impianto::composto;
};

/* ------------------------------------------------------------ */
// --------- Elemento_elettrico
// * generico elemento elettrico
/* ------------------------------------------------------------ */
interface Elemento_elettrico : Elemento
( extent Elementi_elettrici
)
{
  relationship Circuito alimentato
       inverse Circuito::alimenta;
  relationship list<Morsetto> possiede
       inverse Morsetto::fa_parte;
  relationship set<Contatto> contatti
       inverse Contatto::fa_parte;
};
```

```
/* ------------------------------------------------------------ */
// --------- Elemento_primario
// * elemento elettrico da cui si preleva l'input all'impianto
/* ------------------------------------------------------------ */
interface Elemento_primario : Elemento_elettrico
( extent Elementi_primari
)
{
  attribute range{0.0,+inf} corrente_assorbita;
  relationship set<Elemento_calcolato> comanda
       inverse Elemento_calcolato::deriva;
  relationship Elemento_ausiliario contenuto_in
       inverse Elemento_ausiliario::contiene;
  relationship Elemento_ausiliario parti_in
       inverse Elemento_ausiliario::contiene_parti_di;
  relationship Elemento_ausiliario collegato_in
       inverse Elemento_ausiliario::collegato;
};

/* ------------------------------------------------------------ */
// --------- Attuatore
// * Attuatore
/* ------------------------------------------------------------ */
interface Attuatore : Elemento_primario
( extent Attuatori
)
{
  attribute real potenza;
  attribute range {0.0,1.0} cosfi;
  attribute integer num_cicli_lavoro_per_ora;
  attribute range {0,24} num_ore_funz_al_giorno;
  attribute range{1,8} categoria_impiego;

/* puo' essere :
1 -> 'DC-20'
2 -> 'DC-21'
3 -> 'DC-22'
4 -> 'DC-23'

5 -> 'AC-20'
6 -> 'AC-21'
```

```
7 -> 'AC-22'
8 -> 'AC-23'
*/
  attribute real coefficiente_contemporaneita;
};
// se tipo_alimentazione='DC'
// allora categoria_impiego in ('DC-20','DC-21', 'DC-22', 'DC-23')
rule Att1 forall X in Attuatore: X.alimentato.tipo_alimentazione="DC"
     then X.categoria_impiego>1 and X.categoria_impiego<4;

// se tipo_alimentazione='AC'
   allora categoria_impiego in ('AC-20','AC-21', 'AC-22', 'AC-23')
rule Att2 forall X in Attuatore: X.alimentato.tipo_alimentazione="AC"
     then X.categoria_impiego>5 and X.categoria_impiego<8;

// legame tra la corrente e la potenza
rule Att3 forall X in Attuatore: X in Attuatore
     then X.corrente_assorbita=
          range{0.0,100.0} calcola_corrente(X.potenza,
                                            X.cosfi,
                                            X.alimentato.tensione);

/* ------------------------------------------------------------ */
// --------- Sensore
// * Sensore
/* ------------------------------------------------------------ */
interface Sensore : Elemento_primario
( extent Sensori
)
{
  attribute string uscita;
};

/* ------------------------------------------------------------ */
// --------- Elemento_colloquio_uomo
// * Elemento_colloquio_uomo
/* ------------------------------------------------------------ */
interface Elemento_colloquio_uomo : Elemento_primario
( extent Elementi_colloquio_uomo
)
{
```

```
  attribute string uscita;
  attribute real larghezza;
  attribute real altezza;
  attribute real profondita;
};

/* ------------------------------------------------------------ */
// --------- Elemento_calcolato
// * elemento elettrico calcolato in base alle informazioni
// * dei soli elementi primari a cui esso fa capo
/* ------------------------------------------------------------ */
interface Elemento_calcolato : Elemento_elettrico
( extent Elementi_calcolati
)
{
  attribute string tipo_elemento_calcolato;
  relationship set<Elemento_primario> deriva
       inverse Elemento_primario::comanda;
};

/* ------------------------------------------------------------ */
// --------- Sottoelemento
// * elemento elettrico calcolato in base alle informazioni
// * dei soli elementi primari a cui esso fa capo
/* ------------------------------------------------------------ */
interface Sottoelemento : Elemento_calcolato
( extent Sottoelementi
)
{
  attribute range{0.0,+inf} corrente_nominale;
  attribute string categoria_impiego;
  attribute range{0.0,+inf} potere_interruzione;
};

//la categoria_impiego=categoria_impiego degli el. primari associati
rule Sott1 forall X in Sottoelemento: X.tipo_elemento_calcolato="Q"
     then X.categoria_impiego=string categoria_impiego(X.deriva);

//la corrente_nominale=
//somma della corrente_assorbita dagli el primari associati
rule Sott2 forall X in Sottoelemento: X.tipo_elemento_calcolato="Q"
```

```
     then X.corrente_nominale=range {0.0,100.0}
          somma_corrente_nominale(X.deriva,1);

//potere di interruzione e' legato alla categoria di impiego
rule Sott3 forall X in Sottoelemento: X.tipo_elemento_calcolato="Q"
     then X.potere_interruzione= range {0.0,100000.0}
          calcola_potere_interruzione(X.corrente_nominale,X.deriva);

//corrente nominale di un fusibile
rule Sott4 forall X in Sottoelemento: X.tipo_elemento_calcolato="F"
     then X.corrente_nominale=range {0.0,100.0}
          somma_corrente_nominale(X.deriva,1.3);

//la categoria_impiego=categoria_impiego degli el. primari associati
rule Sott5 forall X in Sottoelemento: X.tipo_elemento_calcolato="K"
     then X.categoria_impiego=string categoria_impiego(X.deriva);

//la corrente_nominale=
//somma della corrente_assorbita dagli el primari associati
rule Sott6 forall X in Sottoelemento: X.tipo_elemento_calcolato="K"
     then X.corrente_nominale=range {0.0,100.0}
          somma_corrente_nominale(X.deriva,1);

//potere di interruzione e' legato alla categoria di impiego
rule Sott7 forall X in Sottoelemento: X.tipo_elemento_calcolato="K"
     then X.potere_interruzione=range {0.0,50000.0}
          calcola_potere_interruzione(X.corrente_nominale,X.deriva);

//corrente per un trasformatore
rule Sott8 forall X in Sottoelemento: X.tipo_elemento_calcolato="T"
     then X.corrente_nominale=range {0.0,100.0}
          somma_corrente_nominale(X.deriva,1);

/* ------------------------------------------------------------ */
// --------- Elemento_ausiliario
// * elemento non elettrico, serve da contenitore degli el. elettrici
/* ------------------------------------------------------------ */
interface Elemento_ausiliario : Elemento
( extent Elementi_ausiliari
)
{
```

```
  attribute string tipo_materiale;
  attribute string colore;
  attribute real larghezza;
  attribute real altezza;
  attribute real profondita;
  relationship set<Elemento_primario> contiene
       inverse Elemento_primario::contenuto_in;
  relationship set<Elemento_primario> contiene_parti_di
       inverse Elemento_primario::parti_in;
  relationship set<Elemento_primario> collegato
       inverse Elemento_primario::collegato_in;
};

/* ------------------------------------------------------------ */
// --------- Circuito
// * oggetto astratto che indica i circuiti di alimentazione
// * ha il medesimo comportamento di un elemento primario :
// * possiede sottoelementi, si contiene in un el. ausiliario
// * solo che non possiede sigla  e soprattutto nei suoi calcoli
// * non comanda il circuito, ma bensi tutti gli el primari da lui
// * alimentato
/* ------------------------------------------------------------ */
interface Circuito : Elemento_primario
( extent Circuiti
  keys codice_circuito
)
{
  attribute string tipo_linea;
  // "monofase"
  // "trifase con neutro"
  // "trifase senza neutro"
  attribute string tipo_circuito;
  // "potenza"
  // "ausiliario"
  attribute string tipo_alimentazione;
  // "DC"
  // "AC"
  attribute range{1,6} tensione;
  // 1 significa 6V
  // 2 significa 12V
  // 3 significa 24V
```

```
  // 4 significa 110V
  // 5 significa 220V
  // 6 significa 380V
  attribute range{1,2} frequenza;
  // 1 significa 50Hz
  // 2 significa 60Hz
  relationship set<Elemento_elettrico> alimenta
       inverse Elemento_elettrico::alimentato;
// * questa relazione contempla tutti gli elementi primari alimentati
// * e tutti i sottocircuiti alimentati
};
//tipo alimentazione='DC' -> determinate tensioni
rule Cir1 forall X in Circuito: X.tipo_alimentazione="DC"
  then X.tensione>1 and X.tensione<3;

//tipo alimentazione='AC' -> determinate tensioni
rule Cir2 forall X in Circuito: X.tipo_alimentazione="AC"
  then X.tensione>4 and X.tensione<6;

//calcolo della corrente assorbita
rule Cir3 forall X in Circuito: X in Circuito
     then X.corrente_assorbita=range {0.0,100.0}
          calcola_corrente_circuito(X.alimenta);

/* ------------------------------------------------------------ */
// --------- Filo
// * un filo e' come un Sottoelemento
/* ------------------------------------------------------------ */
interface Filo : Elemento_calcolato
( extent Fili
)
{
  attribute range{1,2} tipo_filo;
   // 1 significa 'di fase'
   // 2 significa 'di neutro'
  attribute range{0.0,+inf} sezione;
  attribute range{0.0,+inf} corrente_impiego;
};
//la corrente_impiego=
  somma della corrente assorbita dagli elementi primari associati
rule Fil1 forall X in Filo: X in Filo;
```

```
     then X.corrente_impiego=range{0.0,100}
          somma_corrente_impiego(X.deriva);

//se il circuito e' di potenza la sezione e' la corrente_impiego/2.5
rule Fil2 forall X in Filo: X.alimentato.tipo_circuito="potenza"
     then X.sezione=range{0.0,50} dividi(X.corrente_impiego,2.5);

//se il circuito e' ausiliario la sezione e' 1,5
rule Fil3 forall X in Filo: X.alimentato.tipo_circuito="ausiliario"
     then X.sezione=(real)1.5;

/* ------------------------------------------------------------ */
// --------- Morsetto
// * un morsetto e' come un Sottoelemento
/* ------------------------------------------------------------ */
interface Morsetto: Elemento_calcolato
( extent Morsetti
)
{
  attribute range{0.0,+inf} sezione;
  attribute range{0.0,+inf} corrente_impiego;
  relationship Elemento_elettrico fa_parte
 inverse Elemento_elettrico::possiede;
};
//la corrente_impiego=
//somma della corrente assorbita dagli elementi primari associati
rule Mors1 forall X in Morsetto: X in Morsetto
     then X.corrente_impiego=range {0.0,100}
          somma_corrente_impiego(X.deriva);

//se il circuito e' di potenza la sezione e' la corrente_impiego/2.5
rule Mors2 forall X in Morsetto: X.alimentato.tipo_circuito="potenza"
     then X.sezione=range{0.0,50} dividi(X.corrente_impiego,2.5);

//se il circuito e' ausiliario la sezione e' 1,5
rule Mors3 forall X in Morsetto: X.alimentato.tipo_circuito="ausiliario"
     then X.sezione=(real)1.5;

/* ------------------------------------------------------------ */
// --------- Contatto
// * un Contatto e' un elemento associato ad ogni Sottoelemento
```

```
/* ------------------------------------------------------------ */
interface Contatto: Elemento_calcolato
( extent Contatti
)
{
  attribute range{1,4} tipo_contatto;
  // 1 significa istantaneo NO
  // 2 significa istantaneo NC
  // 3 significa ritardato NO
  // 4 significa ritardato NC
  relationship Elemento_elettrico fa_parte
       inverse Elemento_elettrico::contatti;
};
```

A.2 Schema OCDL

```
prim Contatto =
  Elemento_calcolato &
  ^ [ tipo_contatto : range 1 4 ,  fa_parte : Elemento_elettrico  ] ;

prim Morsetto =
  Elemento_calcolato &
  ^ [ sezione : range 0.0 +inf , corrente_impiego : range 0.0 +inf ,
      fa_parte : Elemento_elettrico  ] ;

prim Filo =
  Elemento_calcolato &
  ^ [ tipo_filo : range 1 2 , sezione : range 0.0 +inf ,
      corrente_impiego : range 0.0 +inf ] ;

prim Circuito =
  Elemento_primario &
  ^ [ tipo_linea : string , tipo_circuito : string ,
      tipo_alimentazione : string , tensione : range 1 6 ,
      frequenza : range 1 2 ,  alimenta : { Elemento_elettrico }  ] ;

prim Elemento_ausiliario =
  Elemento &
  ^ [ tipo_materiale : string , colore : string , larghezza : real ,
      altezza : real , profondita : real ,
      contiene : { Elemento_primario }  ,
      contiene_parti_di : { Elemento_primario }  ,
      collegato : { Elemento_primario }  ] ;

prim Sottoelemento =
  Elemento_calcolato &
  ^ [ corrente_nominale : range 0.0 +inf ,
      categoria_impiego : string  ,
      potere_interruzione : range 0.0  +inf ] ;

prim Elemento_calcolato =
  Elemento_elettrico &
  ^ [ tipo_elemento_calcolato : string ,
      deriva : { Elemento_primario }  ] ;
```

```
prim Elemento_colloquio_uomo =
  Elemento_primario &
  ^ [ uscita : string , larghezza : real , altezza : real ,
      profondita : real ] ;

prim Sensore =
  Elemento_primario &
  ^ [ uscita : string ] ;

prim Attuatore =
  Elemento_primario &
  ^ [ potenza : real , cosfi : range 0.0 1.0 ,
      num_cicli_lavoro_per_ora : integer ,
      num_ore_funz_al_giorno : range 0 24 ,
      categoria_impiego : range 1 8 ,
      coefficiente_contemporaneita :  real ] ;

prim Elemento_primario =
  Elemento_elettrico &
  ^ [ corrente_assorbita : range 0.0 +inf ,
      comanda : { Elemento_calcolato }  ,
      contenuto_in : Elemento_ausiliario  ,
      parti_in : Elemento_ausiliario  ,
      collegato_in : Elemento_ausiliario  ] ;

prim Elemento_elettrico =
  Elemento &
  ^ [  alimentato : Circuito  ,  possiede : { Morsetto }  ,
       contatti : { Contatto }  ] ;

prim Elemento =
  ^ [ codice_elemento : string , casa_costruttrice : string ,
      serie : string , modello : string ,
      codice_commerciale : string  , descrizione_elemento : string ,
      codice_principale_sigla : string , materiale_sigla : string ,
      impianto : Impianto  ] ;

prim Impianto =
  ^ [ codice_impianto : string , descrizione_impianto : string ,
      luogo : string , codice_cliente : string ,
      temperatura_minima : integer , temperatura_massima : integer ,
```

```
      umidita_minima : range 0 100 , umidita_massima : range 0 100 ,
      altitudine : integer ,  composto : { Elemento }  ] ;


antev Mors3a = Morsetto &
 ^ [ alimentato : ^ [ tipo_circuito : vstring "ausiliario" ] ] ;
consv Mors3c = Morsetto & ^ [ sezione : vreal 1.5 ] ;

antev Mors2a = Morsetto &
 ^ [ alimentato : ^ [ tipo_circuito : vstring "potenza" ] ] ;
consv Mors2c = Morsetto &
 ^ [ sezione : range 0.0 50 function
     dividi ( in corrente_impiego : range 0.0 +inf , in 2.5 : real ) ] ;

antev Mors1a = Morsetto & Morsetto ;
consv Mors1c = Morsetto &
 ^ [ corrente_impiego : range 0.0 100 function
     somma_corrente_impiego ( in deriva : { Elemento_primario } ) ] ;

antev Fil3a = Filo &
 ^ [ alimentato : ^ [ tipo_circuito : vstring "ausiliario" ] ] ;
consv Fil3c = Filo & ^ [ sezione : vreal 1.5 ] ;

antev Fil2a = Filo &
 ^ [ alimentato : ^ [ tipo_circuito : vstring "potenza" ] ] ;
consv Fil2c = Filo &
 ^ [ sezione : range 0.0 50 function
     dividi ( in corrente_impiego : range 0.0 +inf , in 2.5 : real ) ] ;

antev Fil1a = Filo & Filo ;
consv Fil1c = Filo &
 ^ [ corrente_impiego : range 0.0 100 function
     somma_corrente_impiego ( in deriva : { Elemento_primario } ) ] ;

antev Cir3a = Circuito & Circuito ;
consv Cir3c = Circuito &
 ^ [ corrente_assorbita : range 0.0 100.0 function
     calcola_corrente_circuito ( in alimenta : { Elemento_elettrico } ) ] ;

antev Cir2a = Circuito & ^ [ tipo_alimentazione : vstring "AC" ] ;
consv Cir2c = Circuito & ^ [ tensione : range 5 5 ] ;
```

```
antev Cir1a = Circuito & ^ [ tipo_alimentazione : vstring "DC" ] ;
consv Cir1c = Circuito & ^ [ tensione : range 2 2 ] ;

antev Sott8a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "T" ] ;
consv Sott8c = Sottoelemento &
 ^ [ corrente_nominale : range 0.0 100.0 function
     somma_corrente_nominale ( in deriva : { Elemento_primario } ,
                               in 1 : integer ) ] ;

antev Sott7a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "K" ] ;
consv Sott7c = Sottoelemento &
 ^ [ potere_interruzione : range 0.0 50000.0 function
     calcola_potere_interruzione ( in corrente_nominale : range 0.0
 +inf ,
                                   in deriva : { Elemento_primario } )
 ] ;

antev Sott6a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "K" ] ;
consv Sott6c = Sottoelemento &
 ^ [ corrente_nominale : range 0.0 100.0 function
     somma_corrente_nominale ( in deriva : { Elemento_primario } ,
                               in 1 : integer ) ] ;

antev Sott5a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "K" ] ;
consv Sott5c = Sottoelemento &
 ^ [ categoria_impiego : string function
     categoria_impiego ( in deriva : { Elemento_primario } ) ] ;

antev Sott4a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "F" ] ;
consv Sott4c = Sottoelemento &
 ^ [ corrente_nominale : range 0.0 100.0 function
     somma_corrente_nominale ( in deriva : { Elemento_primario } ,
                               in 1.3 : real ) ] ;
antev Sott3a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "Q" ] ;
```

```
consv Sott3c = Sottoelemento &
 ^ [ potere_interruzione : range 0.0 100000.0 function
     calcola_potere_interruzione ( in corrente_nominale : range 0.0
 +inf ,
     in deriva : { Elemento_primario } ) ] ;

antev Sott2a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "Q" ] ;
consv Sott2c = Sottoelemento &
 ^ [ corrente_nominale : range 0.0 100.0 function
     somma_corrente_nominale ( in deriva : { Elemento_primario } ,
                               in 1 : integer ) ] ;
antev Sott1a = Sottoelemento &
 ^ [ tipo_elemento_calcolato : vstring "Q" ] ;
consv Sott1c = Sottoelemento &
 ^ [ categoria_impiego : string function
     categoria_impiego ( in deriva : { Elemento_primario } ) ] ;

antev Att3a = Attuatore & Attuatore ;
consv Att3c = Attuatore &
 ^ [ corrente_assorbita : range 0.0 100.0 function
     calcola_corrente ( in potenza : real ,
     in cosfi : range 0.0 1.0 , in alimentato.tensione : range 1 6 ) ] ;

antev Att2a = Attuatore &
 ^ [ alimentato : ^ [ tipo_alimentazione : vstring "AC" ] ] ;
consv Att2c = Attuatore & ^ [ categoria_impiego : range 6 7 ] ;

antev Att1a = Attuatore &
 ^ [ alimentato : ^ [ tipo_alimentazione : vstring "DC" ] ] ;
consv Att1c = Attuatore & ^ [ categoria_impiego : range 2 3 ] ;


operation Morsetto = range 0.0 50
          dividi ( in param1 : range 0.0 +inf , in param2 : real ) ;
operation Filo = range 0.0 50
          dividi ( in param1 : range 0.0 +inf , in param2 : real ) ;
operation Sottoelemento = range 0.0 100.0
          somma_corrente_nominale ( in param1 : { Elemento_primario }
          ,
                                    in param2 : integer ) ;
```

```
operation Sottoelemento = range 0.0 50000.0
          calcola_potere_interruzione ( in param1 : range 0.0 +inf ,
                                        in param2 : {
          Elemento_primario } ) ;
operation Sottoelemento = string
          categoria_impiego ( in param1 : { Elemento_primario } ) .
```

Appendice B

Microsoft ODBC

B.1 Presentazione di ODBC

ODBC (Open Database Connectivity) e' una interfaccia a livello di chiamata che consente alle applicazioni di accedere ai dati in un database qualsiasi per il quale esiste un driver ODBC. ODBC, infatti, fornisce un'API che consente ad una applicazione di essere indipendente dal sistema di gestione del database (DBMS). I driver ODBC gestiscono i collegamenti alla fonte dei dati e si usa SQL per selezionare i record dal database.

I componenti di ODBC sono:

- API di ODBC
  Una libreria di chiamate di funzione, un insieme di codici di errore e una sintassi di Structured Query Language (SQL) standard per accedere ai dati sui DBMS.

- Driver Manager di ODBC
  Una libreria a collegamento dinamico (ODBC32.DLL) che carica i driver di database di ODBC per conto di un'applicazione. Questa DLL trasparente per l'applicazione.

- I driver di database di ODBC
  Una o piu' DLL che elaborano le chiamate di funzioni di ODBC per specifici DBMS.

- ODBC Cursor Library
  Una libreria a collegamento dinamico (ODBCCR32.DLL) che risiede

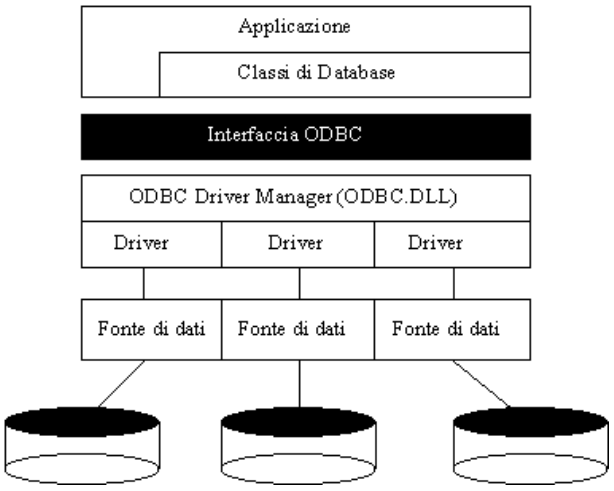

Figura B.1: Architettura di ODBC

tra il Driver Manager di ODBC e i driver di database, essa permette di scorrere attraverso le selezioni effettuate sul database.

- ODBC Administrator
  Uno strumento usato per la configurazione di un DBMS in modo tale da renderlo disponibile come fonte di dati per un'applicazione.

In figura B.1 si evidenzia graficamente la struttura di ODBC.

B.2 Utilizzo di ODBC

Per utilizzare ODBC bisogna seguire la seguente procedura:

- creare la fonte dei dati e renderla visibile al sistema Windows. La fonte dei dati e' un nome logico con il quale le applicazione possono accedere al database. In questo modo le applicazione non sono piu' legate fisicamente al nome del file contenente i dati.
- creare l'applicazione con Visual C++, specificando la necessita del supporto ai database
- collegarsi con la fonte dei dati
- creare una classe derivata dalla CRecordSet, gia' collegata con la fonte dei dati ed eseguire una query su di essa.

- A questo punto il programmatore deve solo manipolare i dati ottenuti dalla classe, oppure puo' sfruttare le funzioni membro offerte dalla classe per fare nuove interrogazioni e operazioni sul database.

B.2.1 La classe CRecordSet

Attraverso la classe CRecordSet e' possibile:

- accedere ad una tabella
- eseguire una interrogazione (SELECT di SQL)
- eseguire un join
- eseguire una procedura gia' memorizzata nel database (CALL)

Quando si crea una classe derivata dalla CRecordSet, si definisce all'interno di essa:

- i dati membri relativi ai campi del record
- i dati membri dei parametri (per interrogazioni parametriche)
- il dato membro m_strFilter contenente la clausola WHERE della SELECT
- il dato membro m_strSort contenente la clausola ORDER BY della SELECT
- il dato membro m_nFields contenente il numero di campi del record
- il dato membro m_nParams contenente il numero di parametri della interrogazione

Una volta creata la connessione ed eseguita l'istruzione SQL e' possibile:

- scorrere tra i record selezionati (MoveFirst, MoveNext, MovePrec, MoveLast, Move)
- controllare l'inizio o la fine dell'insieme di record (IsBOF, IsEOF)
- tornare ad eseguire l'interrogazione (Requery)
- aggiungere, eliminare, modificare record

Appendice C

Normative sugli impianti elettrici

C.1 Siglatura

In questa appendice si intende spiegare il significato delle sigle con le quali si identificano gli elementi sul layout.
Esiste una normativa europea (CEI 44-6, EN 60204-1) che spiega come ogni singolo elemento presente nell'impianto va codificato e identificato.

La sigla si compone di almeno 2 parti:

- '=' codice_principale (o posizione)

- '-' codice_materiale numero progressivo

Il codice principale (chiamato in gergo "posizione") identifica una determinata funzione nell'impianto, quindi tutti gli elementi utili a fare lo spostamento verticale del braccio hanno codice principale "=122".

Il codice materiale identifica il tipo di elemento, cosi tutti i motori si chiamano "-M: seguito poi da un numero progressivo che identifica univocamente l'elemento all'interno della posizione e all'interno della tipologia dell'elemento stesso.

C.1.1 Elenco completo dei codici materiali

Di seguito viene presentato un elenco dei codici dei materiali piu' significativi secondo la normativa CEI 44-6[1]:

	sensori
BF	fotocellula
ST	sensore di temperatura (termostato)
BV	trasd. di velocita'
BP	trasduttore di pressione
BW	trasd. di peso
SL	sensore di livello
SQ	fine corsa
SR	sensore di rotazione
BQ	encoder (o trasduttore di posizione)
SP	sensore di posizione
	attuatori
M	motori
EH	riscaldamento
EL	lampade per illumazione
EV	ventilazione quadri
YB	freno elettromagnetico
YV	elettrovalvola
EC	condizionatore
	elementi di colloquoio con l'uomo
PA	amperometro
P	display
PJ	contatore energia
PS	registratore
PV	voltmetro
HA	segnalatore acustico
HL	segnalatore luminoso
SA	selettore
SH	pulsante luminoso
SB	pulsante
PT	timer
PC	contimpulsi
PF	Frequenzimetro
PN	tachimetro

[1] una trattazione più dettagliata è data in [Mas94]

	dispositivi di protezione
FR	rele' termici per motori
FU	fusibile
FF	Rele' di frequenza
FV	Rele' di tensione
	generatori, alimentatori
GD	generatore CC o alimentatore
GS	generatore CA o alternatore
GC	gruppo elettrogeno
GB	batteria accumulatori
GF	convertitore di frequenza o inverter
	rele', contattori
KM	contattore di potenza
KA	rele' ausiliairi
KT	rele' temporizzato
	apparecchi di manovra per circuiti di potenza
QF	interrutore automatico per circuiti ausiliari
QM	interrutore magnetotermico per motori
QU	sezionatore a fusibili
QS	sezionatore
	trasformatori
TA	trasformatore amporemetrico
TC	trasf. per alimentaz. circuiti ausiliari
TM	trasf. di potenza
TV	trasf. di tensione
TS	stabilizzatore
	Modulatori, convertitori
UA	convertitore di segnale in corrente
UV	convertitore di segnale in tensione
UD	convertitore digitale/analogico
UN	convertitore analogico/digitale
UF	convertitore di frequenza
UJ	convertititore di potenza
	morsetti,prese,spine
XS	presa
XP	spina
XC	connettori
XJ	spinotti di prova
XT	morsetti

C.2 Protezioni

Il grado di protezione di una apparecchiatura elettrica a bassa e media tensione e' definito dalla pubblicazione IEC 144. Esso e' identificato mediante le lettere IP seguite da due cifre caratteristiche. La norma francese NFC 20-010 aggiunge una terza cifra che consente di definire il grado di protezione contro i danni meccanici.

prima cifra - protezione contro corpi solidi	seconda cifra - protezione contro liquidi
0 nessuna protezione	**0** nessuna protezione
1 protetto contro corpi solidi superiori a 50 mm	**1** protetto contro cadute verticali di gocce d'acqua (condensa)
2 protetto contro corpi solidi superiori a 12 mm	**2** protetto contro cadute di gocce fino a 15 gradi dalla verticale
3 protetto contro corpi solidi superiori a 2,5 mm	**3** protetto contro cadute di gocce fino a 60 gradi dalla verticale
4 protetto contro corpi solidi superiori a 1 mm	**4** protetto contro gocce da tutte le direzioni
5 protetto contro la polvere	**5** protetto contro getti d'acqua da tutte le direzioni

C.3 Categorie di impiego

Le categorie di impiego degli apparecchi di manovra[2] sono stabilite dalle normative DIN VDE 0660, parte 102,107 e 200.

C.3.1 Corrente alternata

categoria di impiego	esempi di applicazioni tipiche
AC-1	carichi debolmente induttivi, forni a resistenza
AC-15	comando di carichi elettromagnetici (superiori a 72 VA)
AC-2	motori ad anello: avviamento, arresto
AC-20	inserzione e disinserzione senza carico
AC-21	inserzione e disinserzione di carici ohmici
AC-22	inserzione e disinserzione di carici misti ohmici e induttivi
AC-23	inserzione e disinserzione di motori
AC-3	motori a gabbia: avviamento, arresto
AC-3	manovra ad impulsi, frenatura in controcorrente
AC-5a	comando di lampade a scarica
AC-5b	comando di lampade a incandescenza
AC-6a	comando di trasformatori
AC-6b	comando di batterie di condensatori

C.3.2 Corrente continua

categoria di impiego	esempi di applicazioni tipiche
DC-1	carichi debolmente induttivi, forni a resistenza
DC-13	comando di elettromagneti
DC-20	inserzione e disinserzione senza carico
DC-21	inserzione e disinserzione di carici ohmici
DC-22	inserzione e disinserzione di carici misti ohmici e induttivi
DC-23	inserzione e disinserzione di carichi fortemente induttivi
DC-3	motori con eccitazione in derivazione: inserzione, frenatura
DC-5	motori con eccitazione in serie: inserzione, frenatura
DC-6	comando di lamapade a incandescenza

[2]una trattazione più dettagliata è data in [Sie93]

Appendice D

Lex & Yacc

Lex e *Yacc* sono due utility, molto usate in ambiente UNIX, per la realizzazione di analizzatori sintattici. Di seguito è riportata una breve descrizione dei due programmi.
In realtà in questa tesi sono stati utilizzati altri due programmi *Flex&Bison*, diversi ma compatibili con Lex&Yacc.
Flex e Bison sono due strumenti software che facilitano la scrittura di programmi in linguaggio C per l'analisi e l'interpretazione di sequenze di caratteri che costituiscono un dato testo sorgente.
Entrambi questi strumenti, partendo da opportuni file di specifica, generano direttamente il codice in linguaggio C, che può quindi essere trattato allo stesso modo degli altri moduli sorgenti di un programma.

Flex

Flex legge un file di specifica che contiene delle espressioni regolari per il riconoscimento dei token (componenti elementari di un linguaggio) e genera una funzione, chiamata **yylex()**, che effettua l'analisi lessicale del testo sorgente.

La funzione generata estrae i caratteri in sequenza dal fusso di input. Ogni volta che un gruppo di caratteri soddisfa una delle espressioni regolari viene riconosciuto un token e, di conseguenza, viene invocata una determinata azione, definita opportunamente dal programmatore.
Tipicamente l'azione non fa altro che rendere disponibile il token identificato al riconoscitore sintattico. Per spiegare meglio il meccanismo di funzionamento ricorriamo ad un esempio: l'individuazione, nel testo sorgente, di un numero intero

```
[0-9]+            {
                    sscanf( yytext, "%d", &yylval );
                    return( INTEGER );
                  }
```

l'espressione regolare `[0-9]+` rappresenta una sequenza di una o più cifre comprese nell'intervallo `0-9`. La parte compresa tra parentesi {...} specifica invece, in linguaggio C, l'azione che deve essere eseguita.
In questo caso viene restituito al parser il token `INTEGER` poiché è stato riconosciuto un numero intero.

Bison

Bison è un programma in grado di generare un parser in linguaggio C partendo da un file di specifica che definisce un insieme di regole grammaticali.

In particolare Bison genera una funzione, chiamata **yyparse()**, che interpreta una sequenza di token e riconosce la sintassi definita nel file di input.
La sequenza di token può essere generata da un qualunque analizzatore lessicale; di solito però Bison viene utilizzato congiuntamente a Flex.

Il vantaggio principale che deriva dall'utilizzo di Bison è la possibilità di ottenere un vero e proprio parser semplicemente definendo, in un apposito file, la sintassi da riconoscere.
Ciò avviene utilizzando una notazione molto simile alla *Bakus-Naur Form* (BNF).
Occorre però notare che i parser generati in questo modo sono in grado di riconoscere soltanto un certo sottoinsieme di grammatiche, dette *non contestuali*. A prima vista ciò potrebbe sembrare una limitazione; in realtà questo tipo di grammatica è in genere sufficiente[1] per definire la sintassi di un linguaggio di programmazione.

Per illustrare meglio il funzionamento di questo software utilizziamo un esempio di un possibile input per Bison:

```
var_declaration:    VAR var_list ':' type_name ';' ;
variable_list:      variable_name |
                    variable_list ',' variable_name ;
variable_name:      STRING ;
type_name:          INTEGER | FLOAT | BOOLEAN ;
```

[1] una trattazione più dettagliata e formale è data in [ASU86, FRJL88]

Ogni regola consiste di un nome, o simbolo non terminale, seguito da una definizione, che presenta a sua volta uno o più simboli terminali o non terminali (ovvero nomi di altre regole).

I simboli terminali, rappresentati nell'esempio in carattere maiuscolo, sono i token ottenuti dal riconoscitore lessicale. Il riconoscimento della grammatica avviene con un procedimento di tipo *bottom-up*[2], includendo ogni regola che viene riconosciuta in regole più generali, fino a raggiungere un particolare simbolo terminale che include tutti gli altri.
A questo punto il testo sorgente è stato completamente riconosciuto e l'analisi sintattica è terminata.

In realtà un parser deve svolgere anche altri compiti, come l'analisi semantica e la generazione del codice. Per questo motivo Bison consente al programmatore di definire un segmento di codice, detto *azione*, per ogni regola grammaticale.
Ogni volta che una regola viene riconosciuta il parser invoca l'azione corrispondente, permettendo, ad esempio, di inserire i nomi delle variabili nella symbol table durante l'analisi della sezione dichiarativa di un linguaggio:

```
var_declaration:     VAR var_list ':' type_name ';'
                     {
                         Push( $2 );
                     }
                     ;
```

Nell'esempio illustrato `Push()` è una funzione in linguaggio C che si occupa di inserire una lista di variabili nella `symbol table`.
Il codice che si occupa della traduzione vera e propria può allora essere integrato nel parser attraverso il meccanismo delle azioni semantiche.

[2]descritto ampiamente in [MB91]

Appendice E

Modelli di ingegneria del software

E.1 Il modello DFD

Il modello DFD è uno strumento dell'ingegneria del software utilizzato in una delle prime fasi della progettazione per descrivere le funzionalità del prodotto che si deve sviluppare. Il DFD appartiene alla categoria dei modelli *semi-formali*, ovvero sistemi di rappresentazione basati su costrutti grafici che permettono di schematizzare, in modo non ambiguo, solo una certa parte della conoscenza acquisita. La parte restante viene invece descritta in modo informale oppure utilizzando un metodo di descrizione alternativo. Il modello DFD, in particolare, utilizza un formalismo estremamente semplice ed efficace, permettendo di dettagliare la rappresentazione, in fasi successive, fino a raggiungere il livello desiderato. D'altro canto non offre però alcuna descrizione delle strutture dati e non è in grado di rappresentare l'ordine di esecuzione delle funzioni schematizzate.

Gli elementi grafici del modello DFD sono: gli agenti esterni, i flussi di dati, i processi e i depositi di dati. In figura E.1 sono visualizzati i simboli utilizzati.

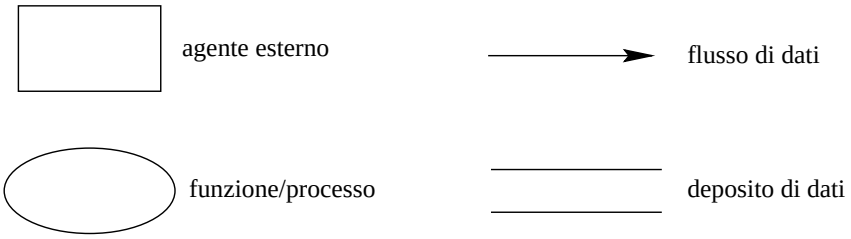

Figura E.1: Simboli del modello DFD.

E.2 Il modello PHOS

Brevemente, riassumiamo le principali regole PHOS per la rappresentazione dei programmi e delle procedure:

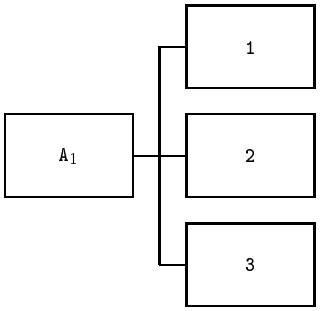

Figura E.2: Esempio di chiamate in sequenza

- La figura E.2 indica che la componente $\mathtt{A}_1$ (ovvero la `procedura` $\mathtt{A}_1$`, indicata con questo tipo di carattere`, o il Modulo A_1) chiama in successione le componenti `1, 2, 3`.

- – nel primo caso della figura E.3 la componente $\mathtt{A}_2$ chiama la componente `1` se si verifica la condizione, altrimenti chiama la componente `2`;
  – nel secondo caso della figura E.3, invece, la componente $\mathtt{A}'_2$ effettua chiamate in alternativa senza che la condizione venga indicata perché è implicita nel nome delle componenti chiamate.

- La figura E.4 indica che la componente $\mathtt{A}_3$ itera su se stessa e ad ogni iterazione chiama la componente $\mathtt{A}'_3$ (ad esempio, riceve in input una

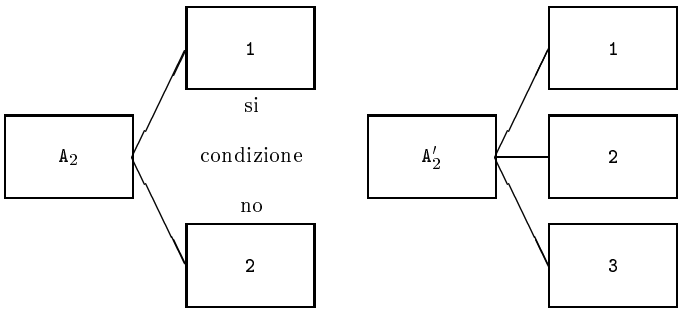

Figura E.3: Esempio di chiamate in alternativa

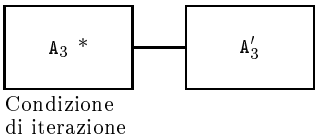

Figura E.4: Esempio di iterazione

lista di elementi da passare singolarmente alla procedura chiamata e quindi dopo aver estratto il primo elemento, itera sul resto della lista finché questa non è vuota). A volte, per chiarezza, viene indicata anche la condizione di iterazione.

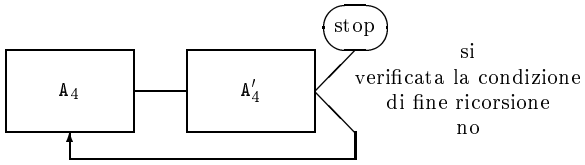

Figura E.5: Esempio di ricorsione

- La figura E.5 indica la ricorsione della componente A_4.

Bibliografia

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[ADF+94] Tom Atwood, Jashua Duhl, Guy Ferran, Mary Loomi, and Dew Wade. Object database standard. R.G.G.Cattell, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1986.

[Atz93] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects.* Springer-Verlag: LNCS n. 701, Heidelberg - Germany, 1993.

[BDK96] F. Bancilhon, C. Delobel, and P. Kanellakis. *building an Object-Oriented Database System, the story of O_2.* Morgan Kaufmann Publishers, Inc., 1996.

[BGSV97] Sonia Bergamaschi, Alessandra Garuti, Claudio Sartori, and Alberto Venuta. Object wrapper: An object-oriented interface for relational databases. In *Euromicro97 Proceedings of the 23rd EUROMICRO Conference*, Budapest, 1-4 Settembre 1997.

[BN94] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[BW86] Kim B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages. In *SIGPLAN Notices*, pages 163–172, 1986.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types - Lecture Notes in Computer Science N. 173*, pages 51–67. Springer-Verlag, 1984.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93, Release 1.1.* Morgan Kaufmann Publishers, Inc., 1994.

[Cor97] Alberto Corni. Odb-dsqo un server www per la validazione di schemi di dati ad oggetti e l'ottimizzazione di interrogazioni conforme allo standard odmg-93. Tesi di Laurea, Facoltà di Scienze dell'Ingegneria, dell' Università di Modena, Modena, 1997.

[EM92] Bertino E and L.D. Martino. *Sistemi di Basi di Dati 0rientate agli Oggetti.* Addison∼Wesley Masson, Milano - Italia, 1992.

[For93] Object Request Broker Task Force. The common object request broker: Architecture and specification. *Journal of Applied Intelligence*, pages 185–203, 1993.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler.* The Benjamin/Cumming Publishing Company, Inc., 1988.

[Gar95] Alessandra Garuti. Ocdl-designer: un componente software per il controllo di consistenza di schemi di basi di dati ad oggetti con vincoli di integrita'. Tesi di Laurea, Facolta' di scienze matematiche fisiche e naturali, corso di laurea di scienze delle informazioni, dell'universita' di Bologna, 1995.

[JBBV95] D. Beneventano J.P. Ballerini, S. Bergamaschi, and M. Vincini. Odb-qoptimizer: un ottimizzatore semantico di interrogazioni per oodb. In A. Albano, S. Salerno, F. Arcelli, M. Gaeta, S. Rizzo, and G. Vantini, editors, *Convegno su Sistemi Evoluti per Basi di Dati*, June 1995.

[LR89] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.

[Mas94] Francesco Dal Mas. *Manuale di disegno elettrotecnico per impianti ed equipaggiamenti industriali e civili.* Edizioni CEI, 1994.

[MB91] T. Mason and D. Brown. *Lex & Yacc.* O'Reilly & Associates Inc., 1991.

[Sie93] Siemens. *Manuale della bassa tensione guida alla scelta degli apparecchi di manovra e di protezione per impianti elettrici.* Tecniche nuove, 1993.

Indice analitico